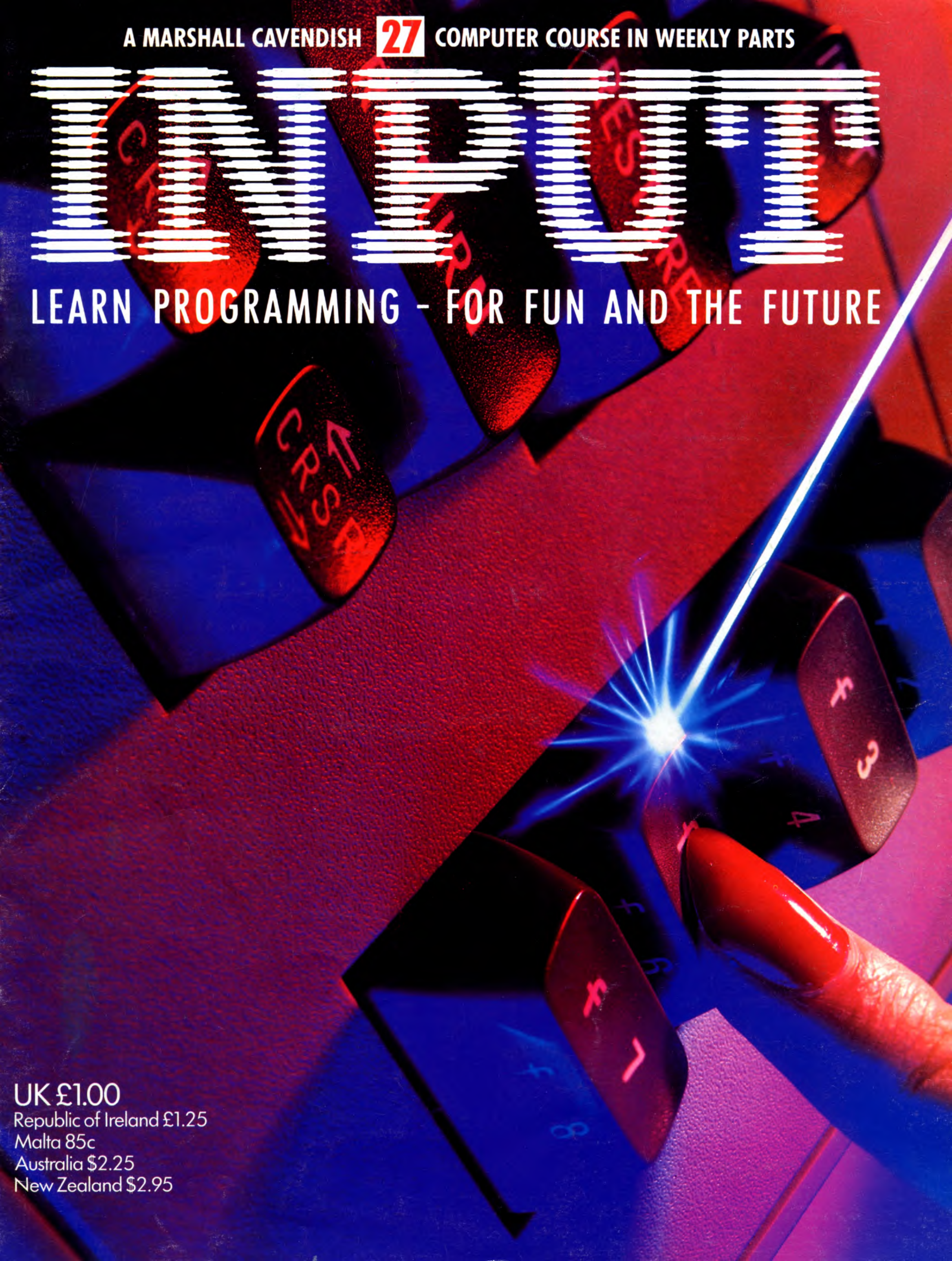
A MARSHALL CAVENDISH **27** COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



UK £1.00
Republic of Ireland £1.25
Malta 85c
Australia $2.25
New Zealand $2.95

INPUT

Vol. 3 No 27

BASIC PROGRAMMING 58

COMMODORE/ACORN CUSTOM KEYS 825

Save wear and tear on your fingers by programming for the special-purpose function keys

GAMES PROGRAMMING 27

AS GOOD AS GOLD 830

The first of two parts of a complete business strategy game in which you run a mining company

BASIC PROGRAMMING 59

MORE HEADLINE IDEAS 838

Find out how to design another, very versatile set of characters for display use

MACHINE CODE 27

ADDING INSTRUCTIONS TO BASIC 844

Program your computer to respond to your own version of BASIC, as well as the one in ROM

APPLICATIONS 15

FINDING A WAY WITH WORDS 852

Enter this simple text editor program, the easy way to tidy up your typing

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS
Front cover,Dave King. Pages 825, 827, 828, Dave King. Pages 830, 832, 834, Johanne Ryder. Pages 839, 843, Dave King. Pages 844, 846, 850, Kuo Kong Chen. Pages 852, 854, Kevin O'Brien.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.

There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland: Send £4.95 (inc p & p) (IR£5.95) for each binder to the address below:
Marshall Cavendish Services Ltd, Department 980, Newtown Road, Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Times Consultants, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

BACK NUMBERS

Back numbers are supplied at the regular cover price (subject to availability).

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services, Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent.

COPIES BY POST

Our Subscription Department can supply copies to any UK address regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. Send your order, with payment to:

Subscription Department, Marshall Cavendish Services Ltd, Newtown Road, Hove, Sussex BN3 7DN

Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +
COMMODORE 64 and 128
ACORN ELECTRON, BBC B and B+
DRAGON 32 and 64
ZX81
VIC 20
TANDY TRS80 COLOUR COMPUTER

COMMODORE AND ACORN CUSTOM KEYS

- KEY DETECTING
- CODE CONTROL
- TESTING FOR SEVERAL KEYS
- LESS TYPING

Even before the QWERTY keys on your Commodore or Acorn become insufficient for complicated software, the Function Keys let you realize the micro's potential

Few home computer users are experienced typists, so for most people, getting used to a keyboard can be difficult. Realizing this fact, manufacturers design certain simplifying features into their keyboards so you are not frustrated by slow progress.

For example, the Spectrum and Electron let you enter entire keywords, such as PRINT or NEXT, by pressing one or at most three keys. Other computers accept abbreviated keywords—such as the BBC's REN. for RENUMBER and P. for PRINT, or the Commo-

dores' and Dragon's ? for PRINT.

These shortened forms can greatly reduce the amount of typing you need to do, but there is an even more powerful facility given by the Function keys (or [f] keys) on the Commodore and Acorn computers. On the Commodores, the [f] keys store valuable control codes which you can access either directly by pressing the key, or indirectly from within a program. And on the Acorn computers, they have the ability to store not only keywords, but even short programs which you can RUN by merely pressing the [f] key.

Much commercial software exploits the [f] keys as the simplest way to control various program functions. These functions may be quite simple operations—for example, in many Commodore games, the function keys set levels, number of players, change playfield colours or start the game—but it is perhaps in business software that they find their most sophisticated use. For example, they may be used to select the various features available in a wordprocessing package—setting markers or editing codes, copying text, etc—freeing the keyboard for entering the text. The BBC even has the facility to label these special keys by inserting a card behind the transparent strip placed above them.

Creative use of the function keys in this way is not limited to commercial packages. Programming them to control your own routines is in most cases a straightforward matter involving simple BASIC commands.

KEY PRESSES

To understand how the [f] keys work, it is useful to know what happens when you press any of the keys on the keyboard.

When this happens the keyboard generates a code which is examined by the built-in ROM software, known as the Operating System. If the code reveals a printable character, such as a letter of the alphabet, this is sent to the screen.

If, however, the keypress reveals a special code for a particular operation, such as [CTRL] 2 (on the Commodores) or [CTRL] L (on the Acorns) or even [RETURN], then an appropriate routine is invoked and the task associated with that key is carried out—the current foreground colour is changed, the screen is cleared, the cursor moves to a new line, or whatever. Similarly, the [f] keys generate a unique code, but the Commodores and Acorns differ in what happens when this code is detected by the Operating System.

On the Commodore and Vic 20, the [f] keys are the four keys set to the right of the QWERTY keyboard. Each can be used with or without [SHIFT], giving a total of eight function keys. Each [f] key has a unique ASCII code, as follows:

[f1] = 133	[f2] ([SHIFT] [f1]) = 137
[f3] = 134	[f4] ([SHIFT] [f3]) = 138
[f5] = 135	[f6] ([SHIFT] [f5]) = 139
[f7] = 136	[f8] ([SHIFT] [f7]) = 140

So ASC([f1]) = 133, or [f1] = CHR$(133), and so on.

When the Operating System detects that an [f] key has been pressed, it does nothing. This is because no code has any meaning or function outside of the one that is written into the operating system. Any of the [f] key codes could have been assigned to a particular letter or to perform some specific action, such as change the screen colour, or move the cursor to the bottom of the screen, but the operating system was designed to ignore the [f] key codes. So it is up to you to make use of them. For example, look at this simple BASIC program:

```
10 GETX$:IF X$="" THEN 10
20 PRINT ASC(X$)
```

When this program is RUN, it stays at Line 10 until a character is typed. But when you press a key, for example, you type A or [RETURN], the program jumps to Line 20, which PRINTs the ASCII code of the character you typed. In a practical application, X$ could be examined to see whether it relates to a particular key. If it does find the code for a given key, the appropriate action can be taken. For example, the program could be directed (using GOTO) to a menu option, or to a subroutine, using GOSUB. Of course, you can specify the code for any key—an A or an L, say. But there is a great advantage in selecting one of the [f] keys—they are separate from the main keyboard and have no other functions until you define them, so there is no possibility of confusion. This is an example of such a simple program.

```
10 GET X$:IF X$="" THEN 10
20 IF ASC(X$)=133 THEN POKE
   53281,(PEEK(53281)+1) AND 15
30 GOTO 10
```

For the Vic 20, change Line 20 as follows:

```
20 IF ASC(X$)=133 THEN POKE
   36879,(PEEK(36879)+16) AND 255
```

Line 20 compares the ASCII code for the character read into the variable X$ with the ASCII code for [f1]; all other keys are ignored. When you RUN the program and type [f1], it changes the screen colour, held in the register at 53281 (36879 on the Vic 20).

Notice that the test at Line 20 could have been written: 20 IF X$=CHR$(133) THEN ... In this case, two *strings* would be tested for equality, instead of two integers as in the previous case. The effect is the same whichever way the program is written.

CODE CONTROL

The Commodore 64 and Vic 20 can store [f] key codes in strings, just as they store codes for control keys, colour change keys, and so on. For example, if you type: WH$="[CTRL] 2" the control character is not acted on immediately, but is stored in the string WH$ as an inverse £ sign ( ). Characters in strings are acted on only when they are printed, by a PRINT statement. Each character is then examined; printable characters are displayed, and control characters are acted on.

The same rules apply to [f] keys, and the effect of the PRINT statement depends on the mode in which the computer is working. In upper case/graphics mode, the [f] keys appear as inverse graphics characters, but in lower case mode, they appear as inverse letters. Here is a list of codes generated by the [f] keys in both modes (the graphic characters for the other keys of the keyboard appear on page 421):

	Upper case/graphics	*Lower case*
[f1]		inverse 'E'
[f3]		inverse 'F'
[f5]		inverse 'G'
[f7]		inverse 'H'
[f2]		inverse 'I'
[f4]		inverse 'J'
[f6]		inverse 'K'
[f8]		inverse 'L'

In the program above, Line 20 could be written: 20 IF X$="[f1]" THEN ... (where [f1] appears as a graphics character or a reversed letter).

TESTING FOR SEVERAL F KEYS

The ASCII codes for each odd-numbered [f] key, followed by those for each even-numbered one are consecutive from 133 to 140. This makes it very easy to write a short routine to detect the code of any one, and act differently in each case:

```
100 GET X$:IF X$="" THEN 100
110 V=ASC(X$)
120 IF V<133 OR V>140 THEN 100
130 ON V-132 GOSUB 1000,3000,5000,
    7000,2000,4000,6000,8000
140 GOTO 10
```

This routine could follow the lines that display a menu. Line 120 ensures that only

the [f] keys can cause the program to branch. If [f1] is pressed, V is assigned the value 133, so V—132 (which equals 1) at Line 13Ø directs the program to the first line number after the GOSUB. Pressing [f1], therefore, is equivalent to GOSUB 1ØØØ. Similarly, if [f2] is pressed, V is assigned the value 137, so V—132 GOSUBs to the fifth line number (2ØØØ), and so on.

UTILITY PACKAGES

Many commercial utility packages allow the user to assign strings to the [f] keys, and you can include a [RETURN] in the string if desired, so that when an [f] key is typed, the assigned string is printed to the screen. [f] keys are normally programmed to reproduce such command strings as PRINT "CLR": LIST, LIST2ØØ-499 or GOTO12ØØ but they may contain any printable or control characters. If the string is terminated with [RETURN], then not only is the string displayed but it is also read and acted on by the system, just as though the user had typed in both the command and [RETURN].

Commands can only be entered into the system while it is in direct or immediate mode: you obviously can't type LIST [RETURN] while a BASIC program is running, and expect the system to execute the LIST command.

This means that you can't write a BASIC program that will assign strings to the [f] keys and make them act as single-keystroke BASIC commands. (Of course, the function keys may be tested for in the usual way inside a program and may be made to trigger any action you like, including printing LIST, say, on the screen, but the system cannot be made to act on this as a command while the program is running).

Software to print command strings and persuade the system to act on them has therefore to be written in machine code and linked into the Operating System software. However it is possible to gain some idea of how this works using BASIC.

When you type [RETURN] on a line containing a command such as LIST, the system processes the string. Now, it doesn't matter how the characters making up that command were put there (whether they were typed in by the user, or printed by a program, for example), the system will take them to mean the LIST command. So if you have assigned "LIST" + CHR$(13) to function key 1, then subsequently typing that function key in direct mode will print LIST on to the screen, execute a return, and perform the LIST operation. But there's something special about the CHR$(13), the [RETURN] character, here. If a program prints out LIST followed by [RETURN], the characters making up the word LIST are printed, and the [RETURN] is printed, moving the cursor to the next line but not causing the LIST to be executed: [RETURN] printed by the computer doesn't have the same effect as [RETURN] entered at the computer keyboard by the user. So the handling of the CHR$(13) in the string "LIST" + CHR$(13) is special: it isn't simply printed to the screen; instead, the system has to be fooled into thinking that the user really has typed [RETURN].

This is quite easy to do, making use of the way the computer handles typed input. BASIC programs are interrupted regularly while they are running to check whether a key has been pressed. The keyboard is scanned 60 times a second by this interrupt-driven routine, and when a key is typed it is detected by this scanning routine and its ASCII code is placed in a buffer at location 631. There is room for 10 characters in this buffer, so the computer can 'remember' up to 10 keystrokes. Memory location 198 stores the number of characters stored in the buffer. So if a program POKEs the ASCII code for [RETURN] into the buffer, and POKEs the value 1 into the location containing current buffer size, you can fool the system into believing that a user has typed [RETURN] at the keyboard. If the program then ends and returns to direct mode, the system will think that [RETURN] has been typed, and it will act on this [RETURN], processing the line the cursor is currently on. This is how many commercial packages handle CHR$(13) characters included in [f] key definitions. The following simple program illustrates this:

```
1Ø PRINT "RUN▢▢▢"
2Ø POKE 631,13: POKE 198,1
```

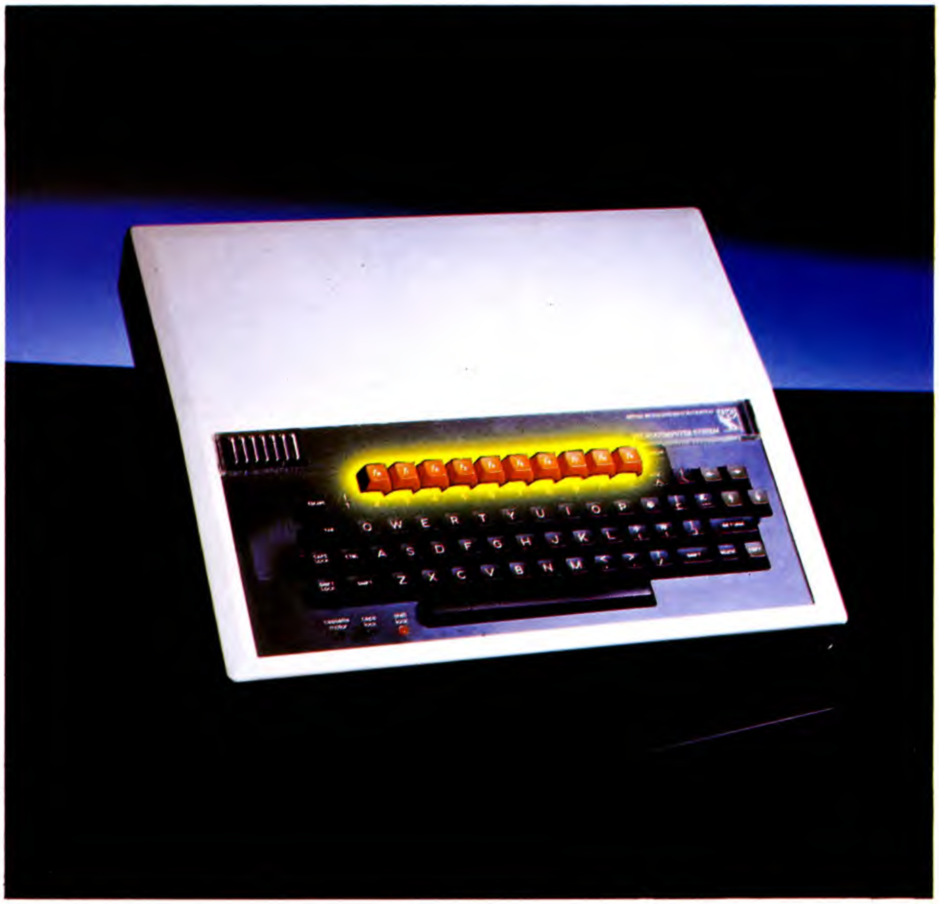
When this is RUN, it prints the string RUN on the screen, followed by three cursor-up characters; it then sets up the input buffer to hold a [RETURN] character, and ends. The system then prints out its READY message, followed by a [RETURN], so the cursor is now on the line containing RUN that was printed by the program. The system now discovers that a [RETURN] is present in its input buffer, so proceeds to read and process the line the cursor is on. Since this says RUN, the program is run again, and the process is repeated indefinitely, the system switching between program mode, where the string containing RUN is printed, and immediate mode, where the [RETURN] in the input buffer is detected and acted on.

The Acorn micros have ten [f] keys, labelled [fØ] to [f9] on the BBC micro and [f1] to [fØ] on the Electron. On the BBC, the red [f] keys are arranged separately at the top of the keyboard, and they are accessed simply by pressing the desired one. On the Electron, they are shared with the numeric keys, so are accessed by pressing [CAPS LK FUNC] first.

The [f] keys are essentially additional to the QWERTY keyboard and are detected in a different way. When you first switch on the computer, the [f] keys are undefined—they are not recognized. So you must define the [f] keys before you can use them. You have a considerable amount of choice over the definition.

PRINTING KEYWORDS

If you are a BBC user, you might set up the [f] keys to store common BASIC keywords, and give single key entries to make light work of entering long programs. A typical line to let you do this looks like:



```
*KEY Ø PRINT
```

Because this line begins with an asterisk (*), it is passed directly to the Operating System, and not to BASIC. Once you have entered it, every time you press [fØ], the word PRINT appears on the screen. Similarly, *KEY 1 THEN would cause the word THEN to appear on the screen every time you press [f1].

These are not necessarily the best keywords to choose to define in this way. In fact, if you are familiar with the abbreviations that BBC BASIC allows, you will be able to enter many keywords using only two or three keystrokes. Many people prefer to use the abbreviations, than to set up the [f] keys at the start of a programming session.

Some keyword abbreviations are not easily remembered and the full word can be awkward. An example is CHR$, which requires the use of [SHIFT], at least once (for $), and often a second time immediately afterwards for an open bracket. The use of long strings as variables, often with upper and lower case letters, can also add to the tedium of entering programs. So if you set up an [f] key with definitions such as 'FNgetposition' or 'PROCsecondstage', you will not only save yourself a lot of typing, but eliminate the risk of mistakes—any difference in spacing or spelling of these variables would cause an error.

You can also enter multi-word definitions, for any expression you are going to use frequently, but you must enclose them in double quotation marks (""). For example, *KEY 3 "IF RND(X) = " would print the string within the quotation marks every time you pressed [f3].

ENTERING RETURN

You can use the [f] keys to print commands on the screen, but the computer will not act upon them unless you enter [RETURN]. So merely setting up [f4], say, to print RENUMBER is not sufficient. Fortunately, there is a control code for [RETURN] that the Operating System understands: this is [CTRL] M. You can enter this code directly, by pressing both keys together, but to include it in a key definition, you must use the key that looks like an elongated colon (¦), the control character. Now the definition for [f4] could look like this:

```
*KEY 4 RENUMBER ¦ M
```

Notice that if a program is running while you press a defined [f] key, the definition is placed in the keyboard buffer to be examined by the Operating System when the program ends. If the definition ends with a control code for RETURN, the computer acts on it. This time, pressing f4 causes RENUMBER to be executed, so the program you are developing is renumbered from 1Ø in steps of 1Ø.

Using this facility, it is a simple step from having the f keys perform single functions to using them for multiple functions or even an entire, short program. A typical example is very useful during program development, when you often need to RUN a program, study the result and make changes. If the program crashes, many of the computer's default values may have been altered so it is difficult or impossible to edit the program. When this happens, you usually have to change mode, select a formatted list option (using LISTO 7), select paged mode, then LIST the program. Here is a key definition to achieve all this:

```
*KEY 5 MODE 1 ¦ M ¦ N LISTO 7 ¦ M LIST
   ¦ M
```

has the same effect as pressing the ¦ N symbol, CTRL and N keys, and mode one. There's a list of all the control codes along with their letters in the User Guide, and you can use any of them in the function key definitions.

Now, every time you press f5, the first page of the program will be listed with a space after each line number, and lines indented within FOR ... NEXT loops and within REPEAT ... UNTIL loops. To scroll the screen to the next page, you need to press SHIFT. Some people find it easier not to use the paged mode, but to press CTRL and SHIFT together to prevent the screen from scrolling, and release these keys when you wish the screen to scroll.

Once you have edited the program, you will need to RUN it, so it is a good idea to set up another f key to achieve this.

You are not just limited to redefining the f keys. One other key that you will find useful to redefine is BREAK. There are many instances when you can press it accidentally, sometimes without knowing. You can't prevent it from BREAKing into the program but you can make sure it will automatically perform an OLD so your program isn't lost. It can be simply defined as:

```
*KEY 1Ø OLD ¦ M
```

There are five other keys as well as the BREAK key that you can redefine. These are COPY (key 11) and the arrow keys (keys 12 to 15). However you first have to enter *FX4,2 which allows them to be reprogrammed. To return them to their normal editing function use *FX4,Ø which resets the keys. And if you want to reset all the function keys use *FX18.

KEY SETTING OPTIONS

The use to which you put the f keys depends on how you use your computer—whether you are developing programs or merely issuing commands in direct mode. Whatever the circumstances, you should aim to use definitions that ease the tedium of typing. Here is a program to set the f keys with some useful definitions:

```
1Ø *KEYØ R. ¦ M
2Ø *KEY1 MO.1 ¦ M ¦ NLISTO7 ¦ MLIST
   ¦ M
3Ø *KEY2 P.~LOMEM–PAGE ¦ M
4Ø *KEY3 CHR$(
5Ø *KEY4 REN. ¦ M
6Ø *KEY5 *LOAD"" 8ØØØ ¦ M
7Ø *KEY6 SAVE
8Ø *KEY7 P.TAB(
9Ø *KEY8 CHAIN"" ¦ M
1ØØ *KEY9
11Ø *KEY1Ø O. ¦ M
12Ø P. "THE KEYS ARE SET. TYPE NEW THEN
    RETURN TO ERASE THIS PROGRAM."
```

When you RUN the program, fØ is set to RUN. Key f1 sets MODE 1 (MO.1) and LISTs any program in memory, as described above. Key f2 prints in hexadecimal the length of a program in memory. Key f5 is an unusual definition and is used as a simple way to verify whether a program has been saved. It attempts to LOAD a program into ROM at memory location 8ØØØ. The program is not actually loaded, but any error messages are printed out so you can check it was SAVEd correctly. Key f8 CHAINs the next program, key f9 is left undefined for you to fill in your own definition, and key f1Ø (BREAK) is redefined as OLD. When you type NEW, as prompted on the screen, this program is erased from memory but the definitions remain, so you can enter your own program without it being corrupted.

It is a good idea to save the function key definitions as a block of memory rather than as a BASIC program, so they can be reloaded 'transparently' without affecting any BASIC program you may be working on. The easiest way to do this is to save the entire function key buffer where the definitions are stored. Use:

```
*SAVE□name□BØØ+1ØØ
```

to save the definitions, and reload them with:

```
*LOAD□name
```

This way you can have several sets of definitions, each saved under a different name, and you can load them in at any time while you are developing or writing a program.

DETECTING THE KEYS

Apart from redefining the f keys, you may simply want to detect when one has been pressed. This is possible on the BBC as well as the Electron. Normally, when you first switch on the BBC, and before any of the keys have been defined, pressing them has no effect whatsoever. Nothing is printed on the screen and no ASCII code is generated. But there are, in fact, five ways to detect them. One way is to use the negative INKEY codes, which are listed on page 275 of the BBC User Guide and page 159 of the Electron User Guide. For example this line detects the fØ key:

```
IF INKEY(–33) THEN ...
```

You can add any command you like after THEN or even use a GOSUB or a REPEAT ... UNTIL checkloop to branch to a more complicated subroutine.

The other way to detect the keys is to make them produce ASCII codes which can then be detected in the normal way using GET or INKEY. You can even choose which ASCII codes are produced. The command to use is *FX 225,n where n is the ASCII code produced by fØ. The other keys follow on from this, so f1 produces code n+1 and so on.

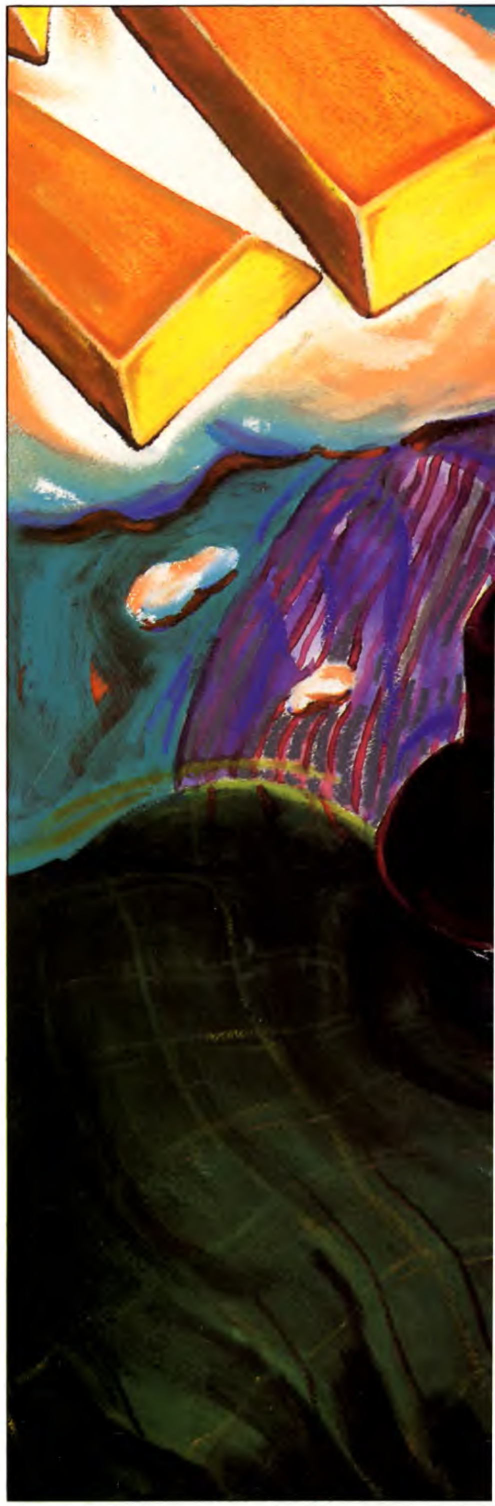
Using other *FX commands—*FX226, *FX227 and *FX228 you can make the keys produce different codes when used with SHIFT, with CTRL or with SHIFT and CTRL together. Again, each command should be followed by a number to determine the code produced by fØ. In fact, when you first switch on, some of these codes are already defined. So, on the BBC, SHIFT plus the f keys generate the Teletext colour control codes from 128 to 137. (There will be more on using Teletext control codes in a later article.) By the way, *FX225,1 returns the keys to their normal user-definable function, and *FX225,Ø makes them have no effect at all.

By using all four *FX commands you can make each key generate four different codes, giving 40 codes in all. This is ideal for programs such as wordprocessors which need to perform a lot of different functions. The red keys can then be used to direct the program to the various subroutines, leaving the normal keyboard free to enter the text.

Luckily, most wordprocessors do still allow you to define the keys in the normal way using the *KEY command, although in this case they must be used with CTRL and SHIFT to print out the definition. The keys are particularly useful in wordprocessing where you can use them to print out a commonly used word or heading.

AS GOOD AS GOLD

Experience the risks and rewards of big business with *INPUT*'s gold mining game. Have you the skill and judgement to make the right decisions and follow them through?



Goldmine is a business strategy game in which you take the part of the owner of a mining company. It is your job to see that the company prospers as well as possible. During the course of the game you are constantly presented with a series of choices—and it's on your ability to make sensible and imaginative decisions that the company fortunes depend.

Strategy games, like adventures, are generally written entirely in BASIC as there is no need for the speed of machine code. And because they do not require lengthy sections of text, it is relatively easy to write them for computers with small memories. Goldmine has been brightened up by the addition of graphics to show the progress of the mine, which add considerably to the memory requirements—but it still fits into the 16K Spectrum, and (with some simplification) into the unexpanded Vic 20.

Even so, the program is quite lengthy, and so it has been split into two parts. In this article, you'll see how to set up the core of the game, but some of the routines which you need to make it playable follow in the next article. When you have entered the listing, SAVE it until next time. On some computers, RUNning the program at this stage presents the player with a screenful of status information, and a series of options. But it will also throw up an error message, as the program is incomplete.

WHAT THE GAME INVOLVES

At the start of the game, you have two assets—the mining company and $2 million in cash. It is your job to invest this wisely in the exploration for the precious metal. The object of the game is to make as much money as possible within 30 turns. You can either play alone, or against an opponent who takes control of a rival concern.

At each turn you are presented with a number of choices. Before you can start mining, you must find a suitable site, so you need to invest in a prospector's report. This will assess your chances of finding gold, its likely depth and the expected amount. It is your job to decide whether the mine is worth exploiting.

Mining is expensive, so you may decide to invest in research and development of new equipment that will lower your costs. Or it may be better to go straight into digging—only you can decide.

If you do start excavations, a graphic display will show you the progress of the mine. If no gold is found, you can elect to continue to dig, or to abandon the mine and start a new working.

During the course of the game two other factors will come into play. When you have found gold, you can store it in your own strongroom, or sell it on the bullion market. It can make sense to keep it, for if you do not need ready cash, it is sensible to keep the gold until the exchange rate is favourable—and the exchange rate fluctuates throughout the game. But be careful, because there are gold robbers about, and the more you have in store, the more tempting the prize.

The second part of this article covers the workings of the game in greater depth. But now, enter the first part of the program.

```
5 BORDER 6: PAPER 6: INK Ø: CLS
1Ø PRINT AT 9,2;"How many players? (1 or
   2)": LET a$=INKEY$: IF a$="" THEN
   GOTO 1Ø
2Ø IF a$<"1" OR a$>"2" THEN GOTO 1Ø
3Ø LET p=VAL a$: LET nop=p
4Ø DIM a(2,6): DIM c(2,5): DIM a$(p,8): DIM
   r(2): LET er=1ØØØØ
5Ø LET r(1)=Ø: LET r(2)=Ø: LET
   a(1,1)=2ØØØØØØ: LET a(1,2)=2ØØØØØØ:
   LET a(2,1)=2ØØØØØØ: LET
   a(2,2)=2ØØØØØØ: LET a(1,3)=Ø: LET
   a(2,3)=Ø: LET a(1,4)=1ØØØØØ: LET
   a(2,4)=1ØØØØØ: LET a(1,5)=Ø: LET
   a(2,5)=Ø: LET a(1,6)=Ø: LET a(2,6)=Ø:
   PRINT
7Ø FOR n=1 TO p: INPUT "Name of
   player□";(n);"?", LINE a$(n): NEXT n
2ØØ FOR n=1 TO 3Ø: FOR m=1 TO nop
2Ø2 BORDER 7: PAPER 7: INK Ø: CLS
21Ø PRINT PAPER 6;n: PRINT PAPER
   1; INK 6;AT Ø,6;"□□G□O
   □L□D□M□I□N□E□□"
22Ø PRINT 'TAB 16;a$(1);: IF nop=2 THEN
   PRINT TAB 24;a$(2);
23Ø PRINT '"TOTAL ASSETS□$";TAB
```

- WRITING A BUSINESS STRATEGY GAME
- WHAT THE GAME INVOLVES
- ONE OR TWO PLAYERS
- THE FIRST SCREEN
- ASSETS AND COSTS
- PROGRESS INFORMATION
- DISPLAYING THE OPTIONS
- THE ROBBERY ROUTINE
- SOME SOUND EFFECTS

```
   15;a(1,1);: IF nop=2 THEN PRINT TAB
   24;a(2,1);
240 PRINT '"CASH ASSETS□ □$"; TAB
   15;a(1,2);: IF nop=2 THEN PRINT TAB
   24;a(2,2);
250 PRINT '"GOLD ASSETS□kg";TAB
   15;a(1,3);: IF nop=2 THEN PRINT TAB
   24;a(2,3);
260 PRINT '"COST TO MINE□$";TAB
   15;a(1,4);: IF nop=2 THEN PRINT TAB
   24;a(2,4);
270 PRINT '"NO. OF MINES";TAB 15;a(1,5);:
   IF nop=2 THEN PRINT TAB 24;a(2,5);
280 PRINT '"MINE DEPTH□ □ □m";TAB
   15;a(1,6);: IF nop=2 THEN PRINT TAB
   24;a(2,6);
300 PRINT '' PAPER 4; INK 0;"Current
   Exchange Rate:—": PRINT "$";er;"□per
   kg of gold"
400 PRINT ' PAPER 5;">—";a$(m)
500 PRINT PAPER 2; INK 7;"1";: PRINT "—
   Research and Development"
510 PRINT PAPER 2; INK 7;"2";: PRINT "—
   Exploration and Report"
520 PRINT PAPER 2; INK 7;"3";: PRINT "—
   Increase mine depth by 200m"
530 PRINT PAPER 2; INK 7;"4";: PRINT "—
   Exchange gold for dollars"
540 PRINT PAPER 2; INK 7;"5";: PRINT "—
   Pass"
550 PRINT : PRINT FLASH 1; PAPER 1; INK
   6;"Enter your instruction"
600 LET i$=INKEY$: IF i$="" THEN GOTO
   600
610 IF i$<"1" OR i$>"5" THEN GOTO
   600
620 GOSUB VAL i$*1000
700 IF a(m,2)<0 THEN GOTO 7000
710 LET er=er+INT (RND*1000)-200
720 IF INT (RND*1600)-a(m,3)<0 THEN
   GOSUB 900
740 LET a(m,1)=a(m,2)+a(m,3)*er
750 PAPER 7: INK 0: BORDER 7: CLS
790 NEXT m
800 NEXT n
810 PAPER 5: BORDER 5: INK 0: CLS
820 PRINT FLASH 1; INK 7; PAPER 2;AT
   6,10;"□GAME OVER□"
830 PRINT 'TAB 5;"Total Assets of□";a$(1):
   PRINT TAB 11;"$";a(1,1)
840 IF nop=2 THEN PRINT 'TAB 5;"Total
   Assets of □";a$(2): PRINT TAB
   11;"$";a(2,1)
850 PRINT '' PAPER 2; INK 6; FLASH 1;TAB
   2;"Press any key to play again"
860 IF INKEY$<>"" THEN GOTO 860
870 IF INKEY$="" THEN GOTO 870
880 RUN
900 PAPER 2: INK 6: BORDER 2: CLS
905 LET jk=INT (RND*100)+50: IF
   jk>a(m,3) THEN LET jk=a(m,3)
```

```
910 PRINT PAPER 6; INK 1; FLASH
    1;AT 9,8;"□R□O□B□
    B□E□R□Y□"
920 PRINT : PRINT INK 7;"□□□□□ You
    have had□";jk;"kg of": PRINT
    "□□□□your gold assets stolen": LET
    a(m,3)=a(m,3)-jk: LET
    a(m,1)=a(m,1)-(jk*er)
930 FOR x=1 TO 35: BEEP .05,40: BEEP
    .05,20: NEXT x
940 BORDER 7: PAPER 7: INK 0: CLS :
    RETURN
```

```
1 POKE52,48:POKE56,48:CLR:PRINT
    "■" TAB(3)"DEFINING
    GRAPHICS,PLEASE WAIT . . ."
2 GOSUB60000
3 POKE53272,28
5 POKE53280,7:POKE53281,7:PRINT
    "■■"
10 PRINT"■■■■■■■■
    ■■■";TAB(5);"HOW MANY
    PLAYERS ? (1 OR 2)":GETA$:
    IFA$=""THEN10
20 IFA$<"1"ORA$>"2"THEN10
30 P=VAL(A$):NO=P
40 DIMA(2,6),C(2,5):ER=10000
50 R(1)=0:R(2)=0:A(1,1)=2000000:
    A(1,2)=2000000:A(2,1)=2000000:
    A(2,2)=2000000
52 A(1,3)=0:A(2,3)=0:A(1,4)=
    100000:A(2,4)=100000:A(1,5)=0:
    A(2,5)=0:A(1,6)=0
54 A(2,6)=0:PRINT "■"
70 FORN=1TOP:PRINT"NAME OF
    PLAYER"N;:INPUTA$(N):A$(N)=
    LEFT$(A$(N),10):NEXT
200 FORN=1TO30:FORM=1TONO
202 POKE53280,1:POKE53281,1:
    PRINT"■■";
205 FORF=54272TO54296:POKEF,0:
    NEXT
210 PRINT"■■"N;TAB(10)"■□G□
    O□L□D□M□I□N□E□
    ■■"
220 PRINTTAB(16);A$(1);:IFNO=2
    THENPRINTTAB(28);A$(2);
230 PRINT:PRINT"■■TOTAL ASSETS";
    TAB(15);A(1,1);:IFNO=2THENPRINT
    TAB(27);A(2,1);
240 PRINT:PRINT"CASH ASSETS";
    TAB(15);A(1,2);:IFNO=2THEN
    PRINTTAB(27);A(2,2);
250 PRINT:PRINT"GOLD ASSETS KG ";
    TAB(15);A(1,3);:IFNO=2THEN
    PRINTTAB(27);A(2,3);
260 PRINT:PRINT"COST TO MINE $";
    TAB(15);A(1,4);:IFNO=2 THEN
    PRINTTAB(27);A(2,4);
270 PRINT:PRINT"NO. OF MINES";
    TAB(15);A(1,5);:IFNO=2THEN
    PRINTTAB(27);A(2,5);
280 PRINT:PRINT"MINE DEPTH M";
    TAB(15);A(1,6);:IFNO=2THEN
    PRINTTAB(27);A(2,6);
300 PRINT:PRINT"■■■CURRENT
    EXCHANGE RATE:—":PRINT"£"ER;
    "□PER KG OF GOLD"
400 PRINT "■■>■■■";A$(M)
500 PRINT"■■■1■□—□
    RESEARCH AND DEVELOPMENT"
510 PRINT"■2■□—□EXPLORATION
    AND REPORT"
520 PRINT"■3■□—□INCREASE
    MINE DEPTH BY 200M"
530 PRINT"■4■□—□EXCHANGE
    GOLD FOR DOLLARS"
540 PRINT"■5■□—□PASS"
550 PRINT"■■■ENTER YOUR
    INSTRUCTION":POKE 198,0
600 GETI$:IFI$=""THEN600
610 IFI$<"1"ORI$>"5"THEN600
620 ONVAL(I$)GOSUB1000,2000,
    3000,4000
700 IFA(M,2)<0THEN7000
710 ER=ER+INT(RND(1)*1000)-200
718 J=0:K=0
720 IFINT(RND(1)*1600)-A(M,3)
    <0THENGOSUB900
740 A(M,1)=A(M,2)+A(M,3)*ER
750 POKE53280,1:POKE53281,1:
    PRINT"■■"
790 NEXTM,N
810 POKE53280,3:POKE53281,3:
    PRINT"■■"
820 PRINT"■■■■■■";TAB
    (15);"■■GAME OVER■■■■"
830 PRINT"TOTAL ASSETS OF□";
    A$(1);TAB(18);"□ARE $"A(1,1)
840 IFNO=2THENPRINT"TOTAL
    ASSETS OF□";A$(2);TAB(18);
    "□ARE $"A(1,2)
850 PRINT:PRINT:PRINT"■■■■
    ■■■■■PRESS ANY KEY TO PLAY
    AGAIN"
860 POKE 198,0
870 IFPEEK(197)=64THEN870
880 RUN 3
900 POKE53280,2:POKE53281,2:
    PRINT"■π"
905 JK=INT(RND(1)*100)+50:IF
    JK>A(M,3)THENJK=A(M,3)
910 PRINT"■■■■■■■■";
    TAB(12)"■□R□O□B□
    B□E□R□Y□"
920 PRINTTAB(9)"■■YOU HAVE HAD
    "JK"KG OF":PRINTTAB(7)"YOUR GOLD
    ASSETS STOLEN"
925 A(M,3)=A(M,3)-JK:A(M,1)=
    A(M,1)-(JK*ER)
930 FORF=0TO24:POKE54272+F,0:
    NEXT
940 POKE54286,5:POKE54290,16:
    POKE54275,1:POKE54296,143:
    POKE54278,240
950 POKE54276,65:FR=5389:FOR
    T=1TO150
960 FQ=FR+PEEK(54299)*9.5:
    HF=INT(FQ/256):LF=FQ-HF*256:
    POKE54272,LF
965 POKE54273,HF:NEXT:POKE
    54296,0
970 POKE53280,1:POKE53281,1:
    PRINT"■■":RETURN
```

```
5 POKE36879,25:PRINT"■■":POKE
    36878,15
10 PRINT"■HOW MANY PLAYERS ?":
    PRINT"■■■■(1 OR 2)":GETA$:
    IFA$=""THEN10
20 IFA$<"1"ORA$>"2"THEN10
30 P=VAL(A$):NO=P
40 DIMA(2,6),C(2,5):ER=10000
50 R(1)=0:R(2)=0:A(1,1)=2000000:
    A(1,2)=2000000:A(2,1)=2000000:
    A(2,2)=2000000
52 A(1,3)=0:A(2,3)=0:A(1,4)=
    100000:A(2,4)=100000:A(1,5)=0:
    A(2,5)=0:A(1,6)=0
54 A(2,6)=0:PRINT "■"
```

Q+A

How do I assure the best returns when playing Goldmine?

When the program is completed and you try playing the game, you'll find that it is very like the real world of commerce, and is riddled with uncertainty.

It's therefore very difficult to find a foolproof route to millionaire status. There are a few useful tips that can be given, though. Mining costs can be reduced by investing in research and development, but it's not advisable to invest too much here because you only have 30 goes to make your fortune.

Only start excavating mines that have a good chance of containing gold, and are relatively shallow—but if you keep passing, you'll find that you soon run out of goes.

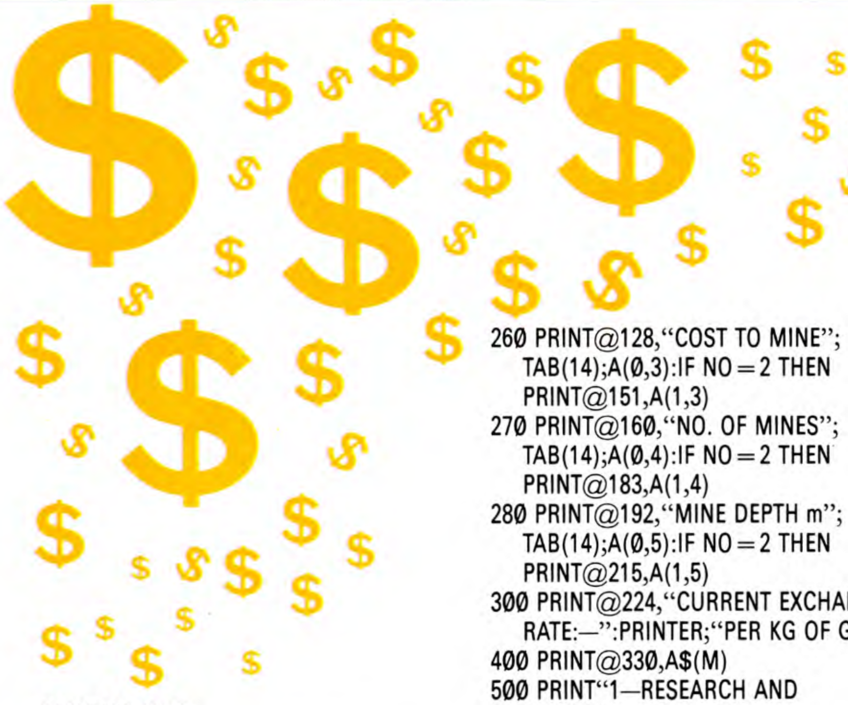
If you are holding any gold, you are in danger of being robbed, but you have to weigh the chances of robbery against the market price of gold—always try to sell at the most favourable price you can get.

```
70 FOR N=1TOP:PRINT"■■■NAME
   OF PLAYER■";N:INPUTA$(N):
   A$(N)=LEFT$(A$(N),9):NEXT
200 FORN=1TO30:FORM=1TONO
210 PRINT"♥■■"N;TAB(4)"■□G□
   O□L□D□M□I□N□E□□■"
220 PRINT"■■"A$(1);:IFNO=2THEN
   PRINTTAB(11);A$(2);
230 PRINT:PRINT"■T"A(1,1);:IF
   NO=2THENPRINTTAB(11);A(2,1)
240 PRINT:PRINT"C"A(1,2);:IFNO=
   2THENPRINTTAB(11);A(2,2)
250 PRINT:PRINT"G"A(1,3);:IFNO=
   2THENPRINTTAB(11);A(2,3);
260 PRINT:PRINT"M"A(1,4);:IFNO=
   2THENPRINTTAB(11);A(2,4);
270 PRINT:PRINT"N"A(1,5);:IFNO=
   2THEN PRINTTAB(11);A(2,5);
280 PRINT:PRINT"D"A(1,6);:IFNO=
   2THENPRINTTAB(11);A(2,6);
300 PRINT:PRINT"■CURRENT EXCHANGE
   RATE:$"ER;"PER KG"
400 PRINT"■■■";A$(M)
500 PRINT"■■1■■RESEARCH,
   DEVELOPMENT"
510 PRINT"■■2■■EXPLORATION,
   REPORT"
520 PRINT"■■3■■INCREASE MINE
   DEPTH□□BY 200M"
530 PRINT"■■4■■EXCHANGE GOLD
   FOR□□□□DOLLARS"
540 PRINT"■■5■■PASS"
550 PRINT"■■ENTER YOUR
   INSTRUCTION":POKE 198,0
600 GETI$:IFI$=""THEN610
610 IFI$<"1"ORI$>"5"THEN600
620 ONVAL(I$)GOSUB1000,2000,
   3000,4000
700 IFA(M,2)<0THEN7000
710 ER=ER+INT(RND(1)*1000)-200
718 J=0:K=0
720 IFINT(RND(1)*1600)-A(M,3)
   <0THENGOSUB900
740 A(M,1)=A(M,2)+A(M,3)*ER
750 PRINT"■♥"
790 NEXTM,N
810 PRINT"■♥"
820 PRINT"■■■■■■■■
   ■■■■■GAME OVER■■■■"
830 PRINT"TOTAL ASSETS OF":
   PRINTA$(1)"ARE $"A(1,1)
840 IFNO=2THENPRINT"TOTAL ASSETS OF
   ":PRINTA$(2)"□ARE $"A(1,2)
850 PRINT"■■■■PRESS ANY KEY TO
   PLAY"
860 POKE198,0
870 IFPEEK(197)=64THEN870
880 RUN
900 PRINT"♥■"
905 JK=INT(RND(1)*100)+50:IFJK
   >A(M,3)THENJK=A(M,3)
910 PRINT"■■■■■■■■■
   ■■■□R□O□B□B□E□R□Y□"
915 FORDE=1TO100:POKE36875,200
   +SIN(DE)*10:NEXTDE:POKE36875,0
920 PRINT"■■■YOU HAVE HAD"
   JK"KG":PRINT"OF YOUR GOLD
   ASSETS■■■STOLEN"
923 FORDE=1TO3000:NEXT
925 A(M,3)=A(M,3)-JK:A(M,1)=
   A(M,1)-(JK*ER)
970 PRINT"♥■":RETURN
```

```
1 MODE1
3 *FX11,0
4 VDU 23;8202;0;0;0;
5 FOR T=224 TO 238:VDU23,T:FOR P=1
   TO 8:READ A:VDU A:NEXT:NEXT
10 VDU 19,2,4,0,0,0,19,0,2,0,0,0:
   PRINTTAB(7,10)"HOW MANY PLAYERS (1
   OR 2)":A$=GET$
20 IF A$<>"1" AND A$<>"2" THEN 10
30 P=VAL(A$):NOP=P
40 DIM A(2,6),C(2,5),A$(P),R(2):
   ER=10000
50 R(1)=0:R(2)=0:A(1,1)=2000000:
   A(1,2)=2000000:A(2,1)=2000000:
   A(2,2)=2000000:A(1,3)=0:A(2,3)=
   0:A(1,4)=100000:A(2,4)=100000:
   A(1,5)=0:A(2,5)=0:A(1,6)=0:
   A(2,6)=0:PRINT
70 FOR N=1 TO P:PRINT"NAME OF PLAYER
   ";N;:INPUTA$(N):NEXT
200 FOR N=1 TO 30:FOR M=1 TO NOP
202 COLOUR131:COLOUR0:CLS
210 COLOUR130:PRINT;N;TAB(10,0)
   "□G□O□L□D□M□I□N□E□":
   COLOUR131
220 PRINTTAB(20,3)A$(1);:IF NOP=2 THEN
   PRINTTAB(30,3)A$(2);
230 PRINT'"TOTAL ASSETS□$"TAB(19)
   A(1,1);:IF NOP=2 THEN PRINTTAB
   (29)A(2,1);
240 PRINT'"CASH ASSETS□□$"TAB(19)
   A(1,2);:IF NOP=2 THEN PRINTTAB
   (29)A(2,2);
250 PRINT'"GOLD ASSETS□□$"TAB(19)
   A(1,3);:IF NOP=2 THEN PRINTTAB
   (29)A(2,3);
260 PRINT'"COST TO MINE□$"TAB(19)
   A(1,4);:IF NOP=2 THEN PRINTTAB
   (29)A(2,4);
270 PRINT'"NO. OF MINES□□"TAB(19
   A(1,5);:IF NOP=2 THEN PRINTTAB
   (29)A(2,5);
280 PRINT'"MINE DEPTH□□□m"TAB
   (19)A(1,6);:IF NOP=2 THEN PRINT
   TAB(29)A(2,6);
300 COLOUR1:PRINT''"CURRENT EXCHANGE
   RATE :—$";ER;"□PER kg OF GOLD"
400 PRINT:COLOUR131:COLOUR2:PRINT
   ">—";A$(M)
500 PRINT'"1□—RESEARCH AND
   DEVELOPMENT"
510 PRINT'"2□—EXPLORATION AND
   REPORT"
520 PRINT'"3□—INCREASE MINE DEPTH BY
   200m"
530 PRINT'"4□—EXCHANGE GOLD FOR
   DOLLARS"
540 PRINT '"5□—PASS"
550PRINT''"ENTER YOUR INSTRUCTION"
600 I$=GET$
610 IF I$<"1" OR I$>"5" THEN 600
620 GOSUB VAL I$*1000
700 IF A(M,2)<0 THEN 7000
710 ER=ER+RND(1000)-200
```


```
720 IF RND(16ØØ) – A(M,3) < Ø THEN
    GOSUB 9ØØ
74Ø A(M,1) = A(M,2) + A(M,3)*ER
75Ø COLOUR131:COLOURØ:CLS
79Ø NEXT
8ØØ NEXT
81Ø CLS
82Ø PRINTTAB(15,12)"GAME OVER"
83Ø PRINT'TAB(5)"TOTAL ASSETS
    OF□"A$(1)"□ARE□$";A(1,1)
84Ø IF NOP = 2 THEN PRINT'TAB(5)
    "TOTAL ASSETS OF□"A$(2)"□ARE
    □$";A(2,1)
85Ø PRINTTAB(Ø,3Ø)"PRESS ANY KEY TO
    PLAY AGAIN"
86Ø G = GET
88Ø RUN
9ØØ COLOUR129:COLOUR3:CLS
9Ø5 JK = RND(1ØØ) + 5Ø:IF JK > A(M,3)
    THEN JK = A(M,3)
91Ø PRINTTAB(12,13)"□R□O□B□
    B□E□R□Y□"
92Ø PRINT''''''"□□□YOU HAVE
    LOST□";JK;"□kg OF YOUR
    GOLD":A(M,3) = A(M,3) – JK:A(M,1)
    = A(M,1) – JK*ER
93Ø FOR X = 1 TO 35:SOUND1, – 15,4Ø,1:
    SOUND1, – 15,1ØØ,1:NEXT
94Ø COLOUR131:COLOURØ:CLS:RETURN
```

```
1Ø PMODE 3,1:CLS
2Ø DIM H(23),T(Ø),D(1),B(1),A(1,5),
    C(1,4),A$(1),R(1)
4Ø FOR K = Ø TO 9:READ NU$(K):NEXT
5Ø DATA NR2D4R2U4BR2,BFEND4BR2,
    R2D2L2D2R2BU4BR2,NR2BD2NR2BD2R2
    U4BR2,D2R2D2U4BR2,NR2D2R2D2L2BE4,
    D4R2U2L2BE2BR2,R2ND4BR2,NR2D4R2U2
    NL2U2BR2,NR2D2R2D2U4BR2
6Ø PCLS3
7Ø DRAW"BM36,23C2L35U6E3R3NU4R5U
    1ØE2RE3R3F4D3F4DF2DF2DF3D2":
    PAINT(18,16),2
8Ø DRAW"BM24,9C3G2D6F5R3E2UH2UHU
    H2UH2BM2Ø,1NLDL2GR5D5BM14,6RBR3
    RBD4DBL4UBD3LBR4RBD2LBL3LBD2RBR
    3RBD2LBL3LBD2RBR3RBD2LBL3L":
```



```
    PAINT(26,15),3
90 DRAW"BM2,21C4UBR4ND3BR4D":
    GET(Ø,Ø)-(37,23),H,G
100 PCLS:DRAW"BM7,ØC4L6BD2ERFRE
    RBD2L7DR7DL5GNR3DNR3FNR4DNR
    4GNR3DNR3FNR4DR2GL3FNR6FR3FL
    4GNRDR5DL3"
110 GET(Ø,Ø)-(7,2),T,G:GET(Ø,3)
    -(7,1Ø),D,G:GET(Ø,11)-(7,2Ø), B,G
120 PRINT@129,"HOW MANY PLAYERS (1
    OR 2)□?";
130 A$=INKEY$:IF A$<"1" OR A$>
    "2" THEN 130
140 P=VAL(A$):NO=P:ER=1ØØØØ:
    A(Ø,Ø)=2ØØØØØØ:A(Ø,1)=2ØØØØØØ:
    A(1,Ø)=2ØØØØØØ:A(1,1)=2ØØØØØØ:
    A(Ø,3)=1ØØØØØ:A(1,3)=1ØØØØØ
150 FORN=1TOP:PRINT:PRINT:PRINT
    "□NAME OF PLAYER";N;:LINE
    INPUTA$(N-1):IF LEN(A$(N-1))
    >8 THEN A$(N-1)=LEFT$(A$
    (N-1),8)
160 NEXT
200 FORN=ØTO29:FORM=ØTONO-1
202 CLS
210 PRINT@3,"goldmine";
220 PRINTTAB(15);A$(Ø);:IF NO=2
    THEN PRINTTAB(24);A$(1)
230 PRINT@32,"TOTAL ASSETS";TAB
    (14);A(Ø,Ø):IF NO=2 THEN PRINT
    @55,A(1,Ø)
240 PRINT@64,"CASH ASSETS";TAB
    (14);A(Ø,1):IF NO=2 THEN PRINT
    @87,A(1,1)
250 PRINT@96,"GOLD ASSETS kg";
    TAB(14);A(Ø,2):IF NO=2 THEN
    PRINT@119,A(1,2)
260 PRINT@128,"COST TO MINE";
    TAB(14);A(Ø,3):IF NO=2 THEN
    PRINT@151,A(1,3)
270 PRINT@16Ø,"NO. OF MINES";
    TAB(14);A(Ø,4):IF NO=2 THEN
    PRINT@183,A(1,4)
280 PRINT@192,"MINE DEPTH m";
    TAB(14);A(Ø,5):IF NO=2 THEN
    PRINT@215,A(1,5)
300 PRINT@224,"CURRENT EXCHANGE
    RATE:—";PRINTER;"PER KG OF GOLD"
400 PRINT@33Ø,A$(M)
500 PRINT"1—RESEARCH AND
    DEVELOPMENT"
510 PRINT"2—EXPLORATION AND REPORT"
520 PRINT"3—INCREASE MINE DEPTH BY
    2ØØm"
530 PRINT"4—EXCHANGE GOLD FOR
    DOLLARS"
540 PRINT"5—PASS";
600 A$=INKEY$:IF A$<"1" OR A$>"5"
    THEN 6ØØ
620 ON VAL(A$) GOSUB 1ØØØ,2ØØØ,
    3ØØØ,4ØØØ,5ØØØ
700 IF A(M,1)<Ø THEN 7ØØØ
710 ER=ER+RND(1ØØØ)-2ØØ
720 IF RND(16ØØ)-A(M,2)<Ø GOSUB 9ØØ
740 A(M,Ø)=A(M,1)+A(M,2)*ER
750 CLS
790 NEXTM,N
810 CLS
820 PRINT@138,"GAME OVER"
830 PRINT@197, "TOTAL ASSETS OF ";
    A$(Ø):PRINTTAB(11);A(Ø,Ø)
840 IF NO=2 THEN PRINTTAB(5);"TOTAL
    ASSETS OF ";A$(1):PRINTTAB(11);A(1,Ø)
850 PRINT@449,"PRESS ANY KEY TO PLAY
    AGAIN"
860 IF INKEY$="" THEN 86Ø ELSE RUN
900 CLS
905 JK=RND(1ØØ)+49:IF JK>A(M,2)
    THEN JK=A(M,2)
910 PRINT@9,"R□O□B□B□E□R□Y"
920 PRINT:PRINT"□□□□□YOU HAVE
    HAD"; JK;"KG OF": PRINT
    "□□□□□YOUR GOLD ASSETS
    STOLEN":A(M,2)=A(M,2)-JK:
    A(M,Ø)=A(M,Ø)-JK*ER
930 PLAY"T4O1CDEFBAGFED"
940 CLS:RETURN
```

All four programs work in much the same way, following the same general structure and line numbering. From the start to line 2ØØ, there are some differences between the programs, though.

To start with, line 1Ø asks for the number of players, and line 2Ø makes sure that your INPUT is within the permitted range. Line 3Ø sets p and nop, according to the chosen number of players.

A series of arrays are DIMensioned in line 4Ø, along with the exchange rate, er. Array a is used for storing information about assets belonging to each of the players and the mines, array c is used for storing information about the mines, array a$ contains the players' names, and array r is used to indicate if mining has started in the mine being considered by the player. Line 5Ø initializes the assets and the mine status for both players. The value Ø is given to r(1) and r(2) to indicate that mining hasn't yet started in the first mine that will be prospected. Other assigned values are given as follows: a(1,1) and a(2,1) are the total assets of each player; a(1,2) and a(2,2) are the cash assets of each player; a(1,3) and a(2,3) are the gold assets; a(1,4) and a(2,4) are mining costs; a(1,5) and a(2,5) are the number of mines; and finally, a(1,6) and a(2,6) are the mine depths. Line 7Ø allows the name of each player to be entered.

In the case of the Commodore 64 program, lines 1 to 5 set up the game's graphics, with the subroutine starting at line 6ØØØØ setting up the mine head workings and the excavation. You need a 3K RAM extension on the Vic 20, and high resolution graphics have been discarded in favour of the machine's block graphics. Line 5 in each of the programs sets up the screen colour and clears the screen. Line 1Ø asks for the number of players, while line 2Ø checks that you have entered one or two. Line 3Ø sets P and NO according to the number of players that has been entered.

Line 4Ø DIMensions two arrays, and sets the exchange rate, ER. Lines 5Ø to 54 initialize the arrays and clear the screen. The values stored in array R tell the program if mining has started in the site being considered by the player—setting the elements equal to zero means that mining hasn't yet begun. The first pair of elements in array A contain each player's total assets, the second pair contain each player's cash assets, the third contain each player's gold assets, the fourth contain the mining costs, the fifth contain the number of mines, and the sixth contain the depth of the current mine. Line 7Ø prompts for the name of the player(s) and stores the response in array A$.

The Acorn program sets MODE 1 in Line 1. Lines 3 and 4 turn off the autorepeat on the keys, and the cursor. Line 5 sets up the graphics for the game by READing from the DATA at the end of the program. Line 1Ø uses a series of VDU commands to set the display colours before prompting for the number of players. Line 2Ø checks the number is within the range allowed. P and NOP are set according to the number of players chosen. A number of arrays are DIMensioned in Line 4Ø, along with the exchange rate, ER. Line 5Ø initializes a range of the elements in these arrays. R(1) and R(2) are set to zero to indicate to the program that excavation hasn't started in the mine being considered by the current player. The first pair of elements in array A contain each player's total assets, the second pair contain the cash assets, the third the gold assets, the fourth, the cost to mine, the fifth, the number of mines and the sixth, the mine depth. Line 7Ø prompts for the name(s) of the player(s).

In the Dragon and Tandy version, PMODE 3 is chosen by Line 1Ø, and the screen is cleared. A series of arrays are DIMensioned in Line 2Ø.

As some of the game takes place on the high resolution graphics screen, text has to be DRAWn at some stages in the game. Lines 4Ø to 11Ø are the routine for drawing numbers on the high resolution screen which you have seen before in *INPUT* (pages 191–192).

Lines 12Ø and 13Ø ask for the number of players and check the number entered. Line 14Ø sets P and NO according to the number of players. Next, some of the elements of array A are initialized; A(Ø,Ø) and A(1,Ø) contain the total assets of each player; the next pair of elements contain the cash assets; the next pair, the gold assets; and the next, the cost to mine. Lines 15Ø and 16Ø prompt for the name(s) of the player(s).

The programs now follow one another very closely. All the programs have a pair of FOR ... NEXT loops starting at Line 2ØØ and finishing at Lines 79Ø and 8ØØ. The loops set up the main menu of options, and the display showing the mining company assets, mining costs, etc.

Variable N (n in the case of the Spectrum) counts the number of goes the player(s) have taken. Variable nop (Spectrum), NO (Commodore, Dragon and Tandy and NOP (Acorn) makes sure that both players get 3Ø turns. Later on in the program, these same variables are used to ensure that both player's assets etc. are displayed.

Line 2Ø2 sets up the screen colours in the case of the Spectrum, Commodore and Acorn. The Dragon and Tandy merely clear the screen. Line 2Ø5 in the Commodore program clears the sound registers ready for sound effects later in the program. In each program, Line 21Ø PRINTs the title: GOLDMINE. Line 22Ø PRINTs the name(s) of the player(s). Only the second name is PRINTed when the two player game has been chosen.

Lines 23Ø to 3ØØ display the values of TOTAL ASSETS, CASH ASSETS, GOLD ASSETS, COST TO MINE, NO. OF MINES, MINE DEPTH and EXCHANGE RATE. If two people are playing, both values are displayed where appropriate by checking the nop, NO or NOP variables. In the Vic 20 program the assets and mine information headings are abbreviated to a single letter owing to the small size of the screen display.

Line 4ØØ displays the name of the player whose turn is in progress. Lines 5ØØ to 54Ø give the player the options Research and Development, Exploration and Report, Increase mine depth by 200 metres, Exchange gold for dollars or Pass. In the Spectrum, Commodore, and Acorn programs, Line 55Ø prompts the player for an instruction. The Dragon/Tandy program doesn't have the prompt because the text screen is full by this stage.

Lines 6ØØ to 62Ø use INKEY$ or GET$ to take in the player's choice, checks that the choice is valid and calls the subroutine which handles that choice.

Line 7ØØ checks if the total assets have dropped below zero, and jumps out the program to the 'end of game' routine, if they have. Line 7ØØØ and those following will be entered next time. Line 71Ø introduces random fluctuations in the exchange rate, so be careful that you sell gold when there's a favourable exchange rate prevailing.

Line 72Ø compares a random number with the amount of gold assets held, to decide if there's going to be a robbery—notice that there is a greater chance of a robbery when you are holding a large amount of gold than when you are holding a small amount. The robbery routine is from Line 9ØØ to 94Ø. Line 9Ø5 chooses how much gold has been stolen, and Line 92Ø displays the amount on screen.

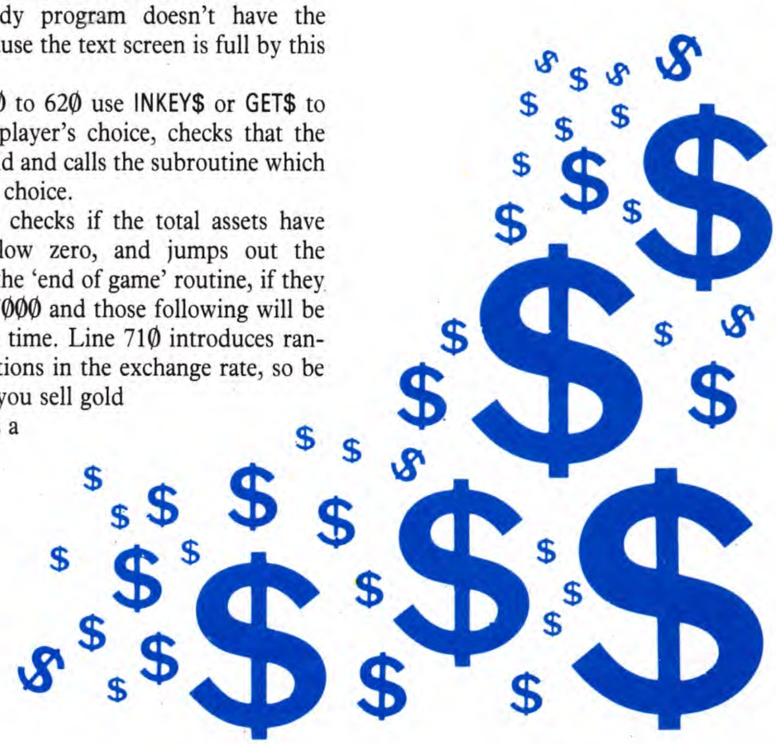
Line 74Ø calculates the total assets by adding the cash assets to the value of the gold held according to the prevailing exchange rate. Line 35Ø resets the screen colours and clears the screen before the NEXTs send the program back to Line 2ØØ ready for the next turn.

Line 81Ø to Line 84Ø are the 'game over' routine, which is used when one of the player's assets drop below zero. The routine displays the financial status of both players, after PRINTing GAME OVER.

Finally, Lines 85Ø to 88Ø are an 'another go?' routine.

Next time you'll be adding a series of subroutines which will make the game playable. There will be a routine which will allow you to reduce your mining costs through research and development, read a report on a prospective mine, excavate the mine in stages, and change gold for dollars.

In addition, there will be all the data you'll need for drawing the graphics illustrating the goldmines, and the progress of excavation.



MORE HEADLINE IDEAS

The first part of this article showed how to create display letters using the character set or block graphics. Here's a new typeface, plus ideas on using all the new letters

Both of the display typefaces which you can create using the programs on pages 815 to 823 are made up from character-square graphics—either based on the standard character set, or made up from the block graphics. But there is another way to create display letters, using high-resolution graphics to DRAW them out, line-by-line.

The programs which follow work in a similar way to those for block graphic letters, in that each consists of a series of DATA statements which tell the computer how to construct each letter. This information is stored in an array, and you can enter the words you want to enlarge in the form of a string. As before, the letters are then built up according to the instructions the computer finds, and displayed on the screen. Later on, you will see how you can then use this display (or that from either of the other letter generators) as part of your own program.

A CUSTOM TYPEFACE

Using high-resolution lines allows you far more freedom over the style of your letters than either of the previous methods. The character set expansion is limited solely to doubling the height or width of the standard characters. And the block graphic letters are completely fixed by their design in the DATA statements.

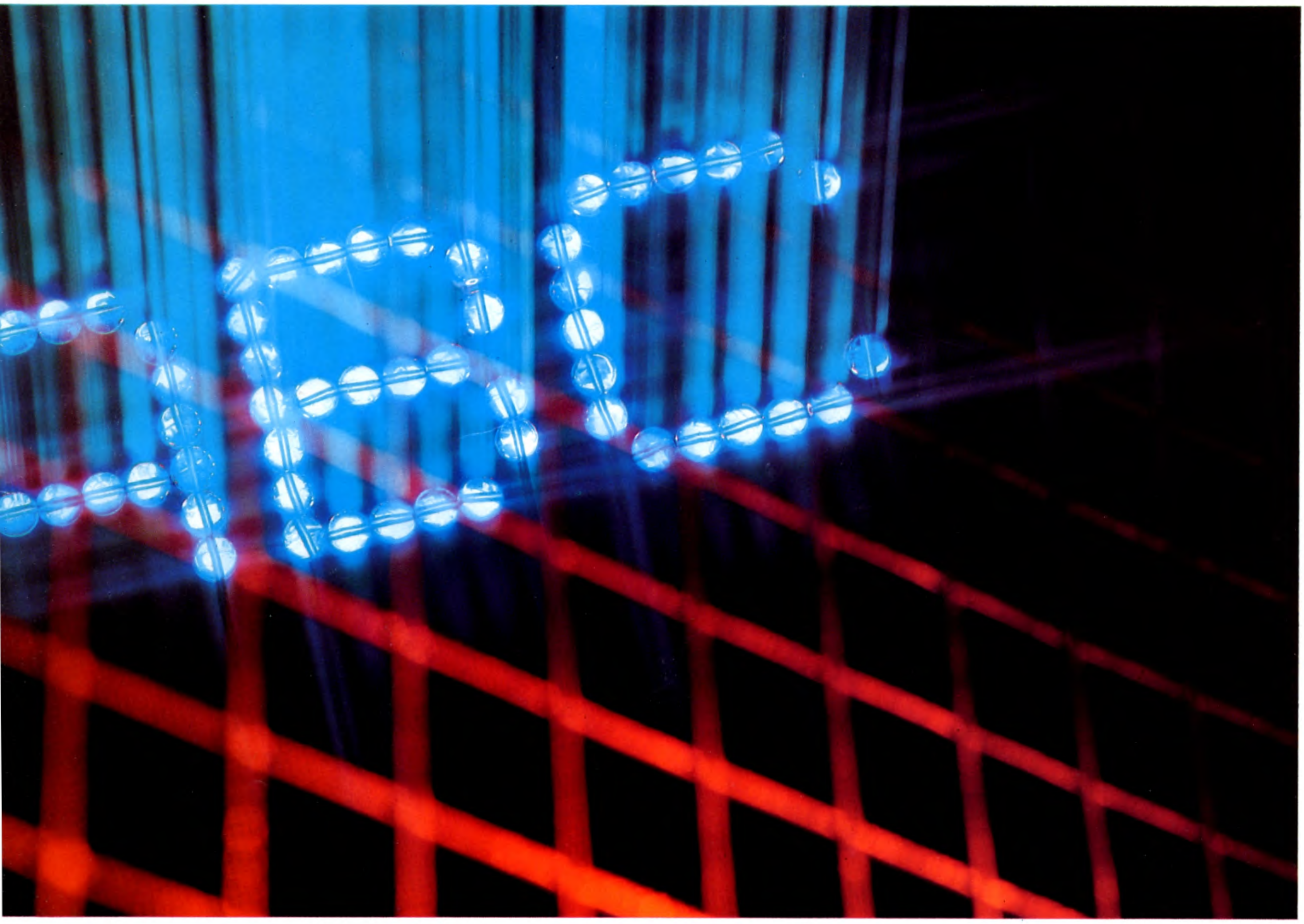
The DATA in the following programs also fixes the way each letter is constructed (so for an L, for example, it tells the computer to DRAW a vertical line and a horizontal line). But all these instructions are relative to each other, and so do not determine the overall shape of the letter. Think of each letter as enclosed by an imaginary box. If the box is tall and thin, you get a tall, thin letter—short and fat, and you get a short, fat letter. The programs are written in such a way that you decide these scaling factors at the outset.

Now type in the program lines. As the program uses up quite a large amount of memory, it will not fit into the Vic 20, and so there is no Vic version.

```
10 POKE 23658,8
20 DIM N(26): DIM A(26,12,2)
30 FOR N=1 TO 26
40 READ N(N)
50 FOR M=1 TO N(N)
60 READ A(N,M,1),A(N,M,2)
70 NEXT M
80 NEXT N
100 INPUT "ENTER A STRING", LINE A$: IF
   A$="" THEN GOTO 100
110 INPUT "ENTER X-FACTOR",X
120 INPUT "ENTER Y-FACTOR",Y
125 CLS
130 FOR N=1 TO LEN A$
140 LET T$=A$(N): IF T$<"A" OR
   T$>"Z" THEN NEXT N: GOTO 100
150 PLOT (10*(N-1)*X)+X*A(CODE
   T$-64,1,1),20+Y*A(CODE T$-64,1,2)
160 FOR M=2 TO N(CODE T$-64)
170 DRAW X*A(CODE T$-64,M,1),
   Y*A(CODE T$-64,M,2)
180 NEXT M
190 NEXT N
200 GOTO 100
1000 DATA 8,0,0,0,5,1,1,4,0,1,-1,0,
   -5,0,3,-6,0
1010 DATA 12,0,0,0,6,5,0,1,-1,0,
   -1,-1,-1,-5,0,5,0,1,-1,0,-1,
   -1,-1,-5,0
1020 DATA 8,6,1,-1,-1,-4,0,-1,1,
   0,4,1,1,4,0,1,-1
1030 DATA 7,0,0,0,6,4,0,2,-2,0,
   -2,-2,-2,-4,0
1040 DATA 7,6,0,-6,0,0,6,6,0,-6,
   0,0,-3,5,0
1050 DATA 6,0,0,0,6,6,0,-6,0,0,
   -3,5,0
1060 DATA 10,5,2,1,0,0,-1,-1,-1,
   -4,0,-1,1,0,4,1,1,4,0,1,-1
1070 DATA 6,0,0,0,6,0,-3,6,0,0,3,
   0,-6
1080 DATA 6,0,0,6,0,-3,0,0,6,-3
   0,6,0
1090 DATA 5,0,1,1,-1,4,0,1,1,0,5
1100 DATA 6,0,0,0,6,0,-3,6,3,-6,
   -3,6,-3
1110 DATA 3,6,0,-6,0,0,6
1120 DATA 5,0,0,0,6,3,-3,3,3,0,-6
1130 DATA 4,0,0,0,6,6,-6,0,6
1140 DATA 9,1,0,-1,1,0,4,1,1,4,0,
   1,-1,0,-4,-1,-1,-4,0
1150 DATA 7,0,0,0,6,5,0,1,-1,0,-1,
   -1,-1,-5,0
```

```
1160 DATA 9,4,2,2,-2,-5,0,-1,1,0,
   4,1,1,4,0,1,-1,0,-5
1170 DATA 9,0,0,0,6,5,0,1,-1,0,-1,
   -1,-1,-5,0,4,0,2,-3
1180 DATA 12,0,1,1,-1,4,0,1,1,0,1,
   -1,1,-4,0,-1,1,0,1,1,1,4,0,1,-1
1190 DATA 4,3,0,0,6,-3,0,6,0
1200 DATA 6,0,6,0,-5,1,-1,4,0,1,
   1,0,5
1210 DATA 3,0,6,3,-6,3,6
1220 DATA 5,0,6,1,-6,2,3,2,-3,1,6
1230 DATA 5,0,0,6,6,-3,-3,-3,3,6,
   -6
1240 DATA 5,3,0,0,3,-3,3,3,-3,3,3
1250 DATA 4,6,0,-6,0,6,6,-6,0
```

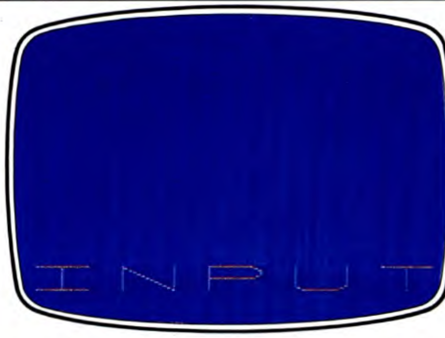
- DRAWING LETTERS LINE BY LINE
- SCALING THE CHARACTERS TO ANY HEIGHT AND WIDTH
- ENTERING THE WORDS
- DESIGN YOUR OWN TYPEFACE
- USING THE LETTERS IN YOUR OWN PROGRAMS
- **SAVE**ING AND **LOAD**ING THE SCREEN DISPLAY



```
5 PRINT"♥INITIALIZING ..."
100 FORT=8192TO16192:POKET,Ø:
   NEXT
110 DIM L(26,2Ø)
115 T=1:X=Ø
140 READA:X=X+1:L(T,X)=A
145 IFA=1ØTHENX=Ø:T=T+1
150 IFT>26THEN16Ø
155 GOTO14Ø
157 PRINT"♥PLEASE WAIT ...":
   FORT=8192TO12ØØØ:POKET,Ø:NEXT
160 X=1Ø:Y=3Ø:PRINT"ENTER
   THE X FACTOR
170 INPUTX1:IFX1>4ØTHEN16Ø
180 PRINT"ENTER THE Y FACTOR"
190 INPUTY1:IFY1>4ØTHEN18Ø
200 PRINT"INPUT THE WORD"
210 INPUTA$
213 FORT=1024TO2Ø23:POKET,1:
   NEXT
215 POKE53265,PEEK(53265)OR32:
   POKE53272,PEEK(53272)OR8
220 FORR=1TOLEN(A$):XT=X
230 S=Ø:A=ASC(MID$(A$,R,1)):A=
   A-64:IFA=-32THENX=X+1Ø:
   XT=X:NEXT
240 S=S+1:B=L(A,S):S=S+1
250 IFB=1ØTHEN73Ø
300 ONBGOTO5ØØ,55Ø,57Ø,59Ø,61Ø,
   63Ø,65Ø,67Ø,73Ø
500 FORM=1TOL(A,S)*Y1
510 Y=Y-1:GOSUB1ØØØ:NEXT: GOTO24Ø
550 FORM=1TOL(A,S)*Y1
560 Y=Y+1:GOSUB1ØØØ:NEXT:
   GOTO24Ø
570 FORM=1TOL(A,S)*X1
580 X=X-1:GOSUB1ØØØ:NEXT:
   GOTO24Ø
590 FORM=1TOL(A,S)*X1
600 X=X+1:GOSUB1ØØØ:NEXT:
   GOTO24Ø
610 FORM=1TOL(A,S)*X1
620 X=X-1:J=(L(A,S)*Y1)/
```

```
(L(A,S)*X1)
622 JJ=J:IF J>1THEN JJ=J/INT(J)
625 FORU=1TOJ:Y=Y-JJ:GOSUB 1000:NEXTU,M:GOTO 240
630 FOR M=1 TO L(A,S)*X1
640 X=X+1:J=(L(A,S)*Y1)/(L(A,S)*X1)
642 JJ=J:IF J>1THEN JJ=J/INT(J)
645 FORU=1TOJ:Y=Y-JJ:GOSUB 1000:NEXTU,M:GOTO 240
650 FOR M=1 TO L(A,S)*X1
660 X=X-1:J=(L(A,S)*Y1)/(L(A,S)*X1)
662 JJ=J:IF J>1THEN JJ=J/INT(J)
665 FORU=1TOJ:Y=Y+JJ:GOSUB 1000:NEXTU,M:GOTO 240
670 FOR M=1 TO L(A,S)*X1
680 X=X+1:J=(L(A,S)*Y1)/(L(A,S)*X1)
682 JJ=J:IF J>1THEN JJ=J/INT(J)
685 FORU=1TOJ:Y=Y+JJ:GOSUB 1000:NEXTU,M:GOTO 240
730 Y=30:X=XT+11*X1:S=0
737 NEXT
740 GETA$:IFA$=""THEN740
750 POKE53265,PEEK(53265)AND223:POKE53272,21
760 GOTO157
1000 RO=INT(Y/8):CH=INT(X/8):LI=YAND7:BI=7-(XAND7):BY=8192+RO*320+CH*8+LI
1010 POKEBY,PEEK(BY)OR2↑BI
1020 RETURN
3000 DATA2,8,1,8,4,8,2,8,1,4,3,8,10
3010 DATA2,8,4,8,1,2,5,2,3,5,4,6,6,2,5,2,3,6,10
3020 DATA4,6,8,2,5,2,3,6,2,8,4,6,6,2,10
3030 DATA2,8,4,6,6,2,1,4,5,2,3,6,10
3040 DATA4,8,3,8,2,4,4,6,3,6,2,4,4,8,10
3050 DATA4,8,3,8,2,4,4,6,3,6,2,4,10
3060 DATA4,7,3,7,2,8,4,8,1,4,3,2,10
3070 DATA2,8,1,4,4,8,1,4,2,8,10
3080 DATA4,8,3,3,2,8,4,4,3,8,10
3090 DATA4,8,3,2,2,6,7,2,3,2,10
3100 DATA2,8,1,4,4,2,6,4,7,4,8,4,10
3110 DATA2,8,4,6,10
3120 DATA2,8,1,8,8,4,6,4,2,8,10
3130 DATA2,8,1,8,8,8,1,8,10
3140 DATA2,8,4,8,1,8,3,8,10
3150 DATA2,8,1,8,4,6,8,2,7,2,3,6,10
3160 DATA2,8,4,8,1,8,3,8,4,8,2,8,8,1,5,4,10
3180 DATA2,8,1,8,4,6,8,2,7,2,3,6,4,3,8,4,10
3190 DATA4,8,3,8,2,4,4,8,2,4,3,8,10
3200 DATA4,8,3,3,2,8,10
3210 DATA2,8,4,8,1,8,10
3220 DATA2,4,8,4,6,4,1,4,10
3230 DATA2,8,6,4,8,4,1,8,10
3240 DATA8,8,5,4,6,4,7,8,10
3250 DATA8,4,6,4,7,8,10
3260 DATA4,8,7,8,4,7,10
```

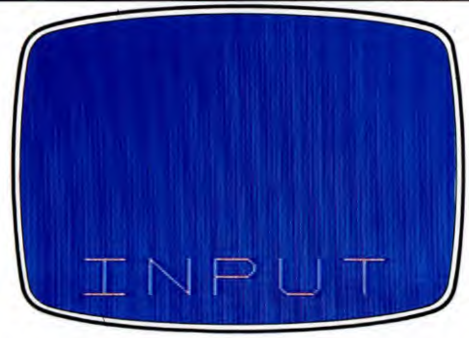
```
10 MODE1
20 DIM N%(26),A%(26,12,2)
30 FOR N=1 TO 26
40 READ N%(N)
50 FOR M=1 TO N%(N)
60 READ A%(N,M,1),A%(N,M,2)
70 NEXT
80 NEXT
100 INPUT"ENTER A STRING",A$:IF A$="" THEN END
110 INPUT"ENTER X-FACTOR",X
120 INPUT"ENTER Y-FACTOR",Y
125 CLS
130 FOR N=1 TO LENA$
140 T$=MID$(A$,N,1):IF T$<"A" OR T$>"Z" THEN NEXT:GOTO 100
150 MOVE (10*(N-1)*X)+X*A%(ASCT$-64,1,1),50+Y*A%(ASC T$-64,1,2)
160 FOR M=2 TO N%(ASCT$-64)
170 PLOT1,X*A%(ASCT$-64,M,1),Y*A%(ASCT$-64,M,2)
180 NEXT
190 NEXT
200 GOTO 100
1000 DATA 8,0,0,0,5,1,1,4,0,1,-1,0,-5,0,3,-6,0
1010 DATA 12,0,0,0,6,5,0,1,-1,0,-1,-1,-1,-5,0,5,0,1,-1,0,-1,-1,-1,-5,0
1020 DATA 8,6,1,-1,-1,-4,0,-1,1,0,4,1,1,4,0,1,-1
1030 DATA 7,0,0,0,6,4,0,2,-2,0,-2,-2,-2,-4,0
1040 DATA 7,6,0,-6,0,0,6,6,0,-6,0,0,-3,5,0
1050 DATA 6,0,0,0,6,6,0,-6,0,0,-3,5,0
1060 DATA 10,5,2,1,0,0,-1,-1,-1,-4,0,-1,1,0,4,1,1,4,0,1,-1
1070 DATA 6,0,0,0,6,0,-3,6,0,0,3,0,-6
1080 DATA 6,0,0,6,0,-3,0,0,6,-3,0,6,0
1090 DATA 5,0,1,1,-1,4,0,1,1,0,5
1100 DATA 6,0,0,0,6,0,-3,6,3,-6,-3,6,-3
1110 DATA 3,6,0,-6,0,0,6
1120 DATA 5,0,0,0,6,3,-3,3,3,0,-6
1130 DATA 4,0,0,0,6,6,-6,0,6
1140 DATA 9,1,0,-1,1,0,4,1,1,4,0,1,-1,0,-4,-1,-1,-4,0
1150 DATA 7,0,0,0,6,5,0,1,-1,0,-1,-1,-1,-5,0
```



Here are three versions of the new typeface on the Spectrum

```
1160 DATA 9,4,2,2,-2,-5,0,-1,1,0,4,1,1,4,0,1,-1,0,-5
1170 DATA 9,0,0,0,6,5,0,1,-1,0,-1,-1,-1,-5,0,4,0,2,-3
1180 DATA 12,0,1,1,-1,4,0,1,1,0,1,-1,1,-4,0,-1,1,0,1,1,1,4,0,1,-1
1190 DATA 4,3,0,0,6,-3,0,6,0
1200 DATA 6,0,6,0,-5,1,-1,4,0,1,1,0,5
1210 DATA 3,0,6,3,-6,3,6
1220 DATA 5,0,6,1,-6,2,3,2,-3,1,6
1230 DATA 5,0,0,6,6,-3,-3,-3,3,6,-6
1240 DATA 5,3,0,0,3,-3,3,3,-3,3,3
1250 DATA 4,6,0,-6,0,6,6,-6,0
```

```
10 PMODE1,1:PCLS
20 DIMN(26),A(26,12,2)
30 FORN=1TO26
40 READN(N)
50 FORM=1TON(N)
60 READA(N,M,1),A(N,M,2)
70 NEXTM,N
80 CLS:INPUT"ENTER A STRING";A$:IFA$=""THENGOTO80
90 INPUT"ENTER X-FACTOR";X
100 INPUT"ENTER Y-FACTOR";Y
110 CLS:PCLS:SCREEN1,0:FORN=1TOLEN(A$)
120 T$=MID$(A$,N,1):IFT$<"A"OR T$>"Z"THENNEXTN:GOTO80
130 J=(10*(N-1)*X)+X*A((ASC(T$)-64),1,1):K=10+Y*A(ASC(T$)-64,1,2)
140 FORM=2TON(ASC(T$)-64)
150 F=X*A((ASC(T$)-64),M,1)
160 G=Y*A((ASC(T$)-64),M,2)
170 LINE(J,K)-(J+F,G+K),PSET
180 J=J+F:K=K+G
190 NEXTM,N
200 IF INKEY$=""THEN 200
210 GOTO80
1000 DATA9,0,6,0,-5,1,-1,4,0,1,1,0,3,0,2,0,-3,-6,0
```



By altering the X and Y factors the letters can be made to any size

These pictures show normal, extra wide and extra tall letters

```
1010 DATA12,0,0,0,6,5,0,1,-1,0,-1
   -1,-1,-5,0,5,0,1,-1,0,-1,
   -1,-1,-5,0
1020 DATA8,6,1,-1,-1,-4,0,-1,1,0,
   4,1,1,4,0,1,-1
1030 DATA7,0,0,0,6,4,0,2,-2,0,-2,
   -2,-2,-4,0
1040 DATA7,6,0,-6,0,0,6,6,0,-6,0,
   0,-3,5,0
1050 DATA7,0,0,6,0,-6,0,0,3,5,0,
   -5,0,0,3
1060 DATA10,7,1,-1,-1,-4,0,-1,1,
   0,4,1,1,4,0,1,-1,0,-1,-1,0
1070 DATA6,0,0,0,6,0,-3,6,0,0,3,
   0,-6
1080 DATA6,0,0,6,0,-3,0,0,6,-3,0,
   6,0
1090 DATA7,0,0,6,0,-3,0,0,5,-1,1,
   -1,0,-1,-1
1100 DATA6,0,0,0,6,0,-3,6,3,-6,-3,
   6,-3
1110 DATA3,0,0,0,6,6,0
1120 DATA5,0,6,0,-6,3,3,3,-3,0,6
1130 DATA4,0,6,0,-6,6,6,0,-6
1140 DATA9,1,0,-1,1,0,4,1,1,4,0,
   1,-1,0,-4,-1,-1,-4,0
1150 DATA7,0,6,0,-6,5,0,1,1,0,1,
   -1,1,-5,0
1160 DATA11,5,0,-4,0,-1,1,0,4,1,
   1,4,0,1,-1,-2,-1,2,1,0,-4,
   -1,-1
1170 DATA9,0,6,0,-6,5,0,1,1,0,1,
   -1,1,-5,0,4,0,2,3
1180 DATA12,6,1,-1,-1,-4,0,-1,1,
   0,1,1,1,4,0,1,1,0,1,-1,1,-4,0,
   -1,-1
1190 DATA4,3,6,0,-6,-3,0,6,0
1200 DATA6,0,0,0,5,1,1,4,0,1,
   -1,0,-5
1210 DATA3,0,0,3,6,3,-6
1220 DATA5,0,0,1,6,2,-3,2,3,1,-6
1230 DATA5,0,0,6,6,-3,-3,-3,3,6,
   -6
1240 DATA5,3,6,0,-3,-3,-3,3,3,3,
   -3
1250 DATA4,0,0,6,0,-6,6,6,0
```

Apart from the Commodore these programs are similar. There are 26 DATA statements, one for each letter of the alphabet. And each one contains a series of numbers which tells your computer how to draw the shape of that letter, as a series of short lines. The first number after each DATA statement is the total number of lines that are used to make up each letter—an L needs fewer than an S, for example. The maximum number used is 12.

The following numbers are arranged in pairs giving the x and y coordinates of each short section of line. As explained above, these numbers are relative, not absolute, so the actual lines that are drawn may be affected by scaling factors. And the first pair of coordinates specify a starting point for the letter within its imaginary 'box'.

The numbers from the DATA statements are READ into two arrays, N and A. A is a three dimensional array, 26 (for the number of letters) by 12 (for the maximum number of lines) by 2 (for the x and y vectors).

ENTERING A STRING

The lines which let you enter your words are the same as those in the program on pages 815 to 823. They check to make sure that you do enter an actual string, rather than a 'null' entry, but there is no limit on the string length you can enter.

The programs then let you INPUT two values—an 'X-factor' and a 'Y-factor'. The values that you enter here determine the actual size of each letter (the dimensions of the invisible 'box'). As a rough guideline to scale, 1 represents a standard size character. 2 would give you double height or double width, 0.5 would give you half height or half width, and so on. Note that for values smaller than one, you may get some odd effects. Since the computer cannot draw a fraction of a pixel, it defaults to drawing a whole pixel. And letters which contain a large number of lines have more opportunities for this to happen. As a result, an S, say, may end up slightly over-sized compared to a letter T.

By having different values for each factor, you can produce some interesting variations on the characters: either by having tall and thin letters, or wide and short ones.

As there is no limit on how long a string you can enter, you should be careful not to enter too many letters for the available screen display at your chosen size—otherwise the computers either stop with an error message or produce strange variations of the letters. You can work out how many characters you can have for any given scale factors quite simply. You should note that the important value is the X-factor which sets how much wider, or narrower, each letter is (the Y-factor determines how tall each letter is). If you have values of 2 as your multiplication factor, you can only fit half as many characters as normal in a line. And if you had a multiplication factor of 4, you could only fit a quarter as many letters onto a line.

ANALYZING THE STRING

As with the other programs which create a display typeface, this one uses a main loop to go through every letter in the string. It starts at Line 130 for the Spectrum, Line 220 for the Commodore, Line 130 for the Acorn, and Line 110 for the Dragon and Tandy.

The program continues in a similar way to the earlier letter generator, setting a string variable (T$) equal to each letter of the string in turn. The Commodore uses a variable A equal to the ASCII code of the letter. The program then checks to see whether the character is a letter from A to Z (any other character is treated as a space). If it is not, the computer does a NEXT to update the FOR ... NEXT loop, and if there are no more letters in your string the computer goes back to let you enter another string.

When the computer finds a letter in the string, it goes to Line 150, which starts the drawing routines. These are at Line 150 for the Spectrum and 170 for the Acorn, Dragon and Tandy. These are slightly different for each computer, since each has a different size of screen and different PLOT and DRAW commands.

The basis for the routine, though, is the same on all the computers except the Commodore which is described later. The control variable of the FOR ... NEXT loop, N, is used as a guide for the x coordinate of the starting position, and the computer adds two things to this. The first is a relative value—the drawing coordinates from the array A—while the second is a value to take into account the screen size and your x and y scaling factors.

You have already seen that one of the three elements of this array is used to separate the details of the x and y coordinates.

The other two elements give each of the 26 letters up to 24 numbers each—remember that 12 is the maximum number of drawn lines used by any letter. The one dimensional array N is used to tell the computer how many lines each letter uses. So, when the computer comes to draw, for example, the letter A, it first looks up how many lines it has to draw from array N. This number then controls the loop to draw the exact number of lines for each letter.

The values of the numbers in A, the main, three-dimensional array, are 64 less than the character codes of each letter—this is the reason why Lines 130 to 180 (Line 230 on the Commodore) contain either the expression ASC(T$)—64 or CODE T$—64 or A—64. What this means is that the computer can take the character code of the letter and uses this to count through to the right part of the arrays to use for drawing the lines. The relative values for x and y stored in the array are then multiplied by the scaling factor that you entered and these are passed to the DRAW or PLOT command.

Since the Commodore cannot DRAW lines on the screen with a single command, the letters have to be built up from dots POKEd onto the screen one at a time by the subroutine at Line 1000. The short routines from Lines 500 to 730 determine the direction of the lines that are drawn—there are eight routines for left, right, down, up and the four diagonals. The DATA statements determine the length of each line.

When the computer has drawn a letter, it goes to the next letter in your string—if there is one—and runs through the same process to draw that. If there are no more letters left to be drawn, the computer returns to Line 100 (Line 80 on the Dragon and Tandy) to let you enter another string. The Dragon and Tandy first wait for you to press a key, since they have to clear the screen in order to return to the text screen (the letters are drawn on the graphics screen).

DESIGNING YOUR OWN

You have now seen three examples of display typefaces but you can use similar ideas to make your own special letters. While you cannot vary the double width and height versions of the computers' own characters, you can use the 'expanding' routines to expand your own redefined characters (the article on pages 450 to 457 explains how you can redefine the character set, except for the Dragon and Tandy, which do not have this facility).

The variations you could exploit with the other two methods, though, are almost endless: you can design your own characters from the computers' block graphics characters, and store them in an array as in the program on pages 815 to 823. If you are not satisfied with the on-board block characters, why not design your own UDGs and combine them? Similarly you could make a variation on the DRAWn typeface in this article—all you have to do is change the DATA, once you have worked out your new characters, and change the array if necessary.

USING LARGE LETTERS

It is all very well being able to create large letters, but unless you can use them in your programs, there is little point. Whether you use the routines given on pages 815 to 823, or the drawing program in this article, you need to be able to call up the special letters for use elsewhere.

There is a number of ways you can use the display typeface in your own programs. The most obvious of these is to incorporate the letter generating programs as a subroutine—then, you can GOSUB them whenever you want to.

If you want to use the expansion of the character set routine, this is probably the best solution. But for the letter-drawing program given here, it is far from ideal. The reason for this is that the program takes up quite a lot of memory space, so that you would not have very much left for your own use.

You can get around this in several ways, all of which involve using the programs to design your letters, then storing the results so that they are available for use elsewhere.

One way of doing this is to store your designs in UDGs. This is easy on the Dragon and Tandy, which can simply GET what is on the screen. On the other computers you could do this by POKEing the UDGs with the contents of the screen. But, except on the Dragon and Tandy, this, too, is complicated.

Another alternative is to SAVE the section of memory which stores the screen, as a block of machine code onto tape. You can then LOAD the display back in to your computer. While it is in your computer, you must protect it by storing it above the end of BASIC, or below the start of BASIC (the article on pages 450 to 457 explains more about this). And once it is in memory, you have to be able to use it. This is not applicable to the Acorn.

The best way of doing this is either to write a routine to PEEK your code in the protected area of RAM, then POKE it into the screen area of memory, or to alter the character set pointers so that your code 'becomes' several characters. You could then PRINT the characters to make your large letters appear. The routine to move your block of code from its protected area in RAM to the screen would look something like this:

```
1000 FOR x=1 TO (the length of your block
     of code)
1010 POKE (the start address of screen
     memory)+x, PEEK (the start address of your
     block of code)+x
1020 NEXT x
```

Again, this routine is not a very good solution to the problem, since it would take quite a long time to finish, unless you know machine code, and can write the equivalent routine in machine code. You could then call the routine and your letters would appear on the screen almost instantly.

SAVEING THE SCREEN

The other alternative is to SAVE the screen picture, and LOAD this in when you LOAD your program. The picture will then stay on the screen until you wipe it off by clearing the screen. On most machines, this is the best option, but the Commodore computers do not have this facility.

The Spectrum has a special keyword which lets you SAVE a screen picture to tape very easily. The picture is SAVEd as a block of memory as described on pages 532 and 533, except that instead of having to type in the start address of the screen and how much memory you wish to SAVE, you can just type the keyword SCREEN$ immediately after the filename. So this command entered as a direct command would SAVE the picture named 'pic':

```
SAVE"pic"SCREEN$
```

To LOAD the screen picture back, use:

```
LOAD" "SCREEN$
```

You can use the display as a title page to a game by LOADing it in first so it stays on the screen while the game is loading. The way to do it is to create a loader program:

```
10 LOAD ""SCREEN$
20 LOAD"GAME"
```

then save this using

```
SAVE"LOADER" LINE 10
```

so that it aoutoruns. The game itself should also be saved using SAVE "GAME" LINE 10 so that it autoruns too. You'll also need to save the programs in the correct order on the tape— LOADER, SCREEN then GAME.

To SAVE a screen picture from the Acorn computers, you should use a command within a program—although you *can* do it as a direct command, you should SAVE it via a program line, since, if you don't, the computer will SAVE any writing on the screen (including the command itself) along with the picture.

The command you should use to SAVE a screen to tape is VDU 21, then:

```
*SAVE"filename"start address□end address
```

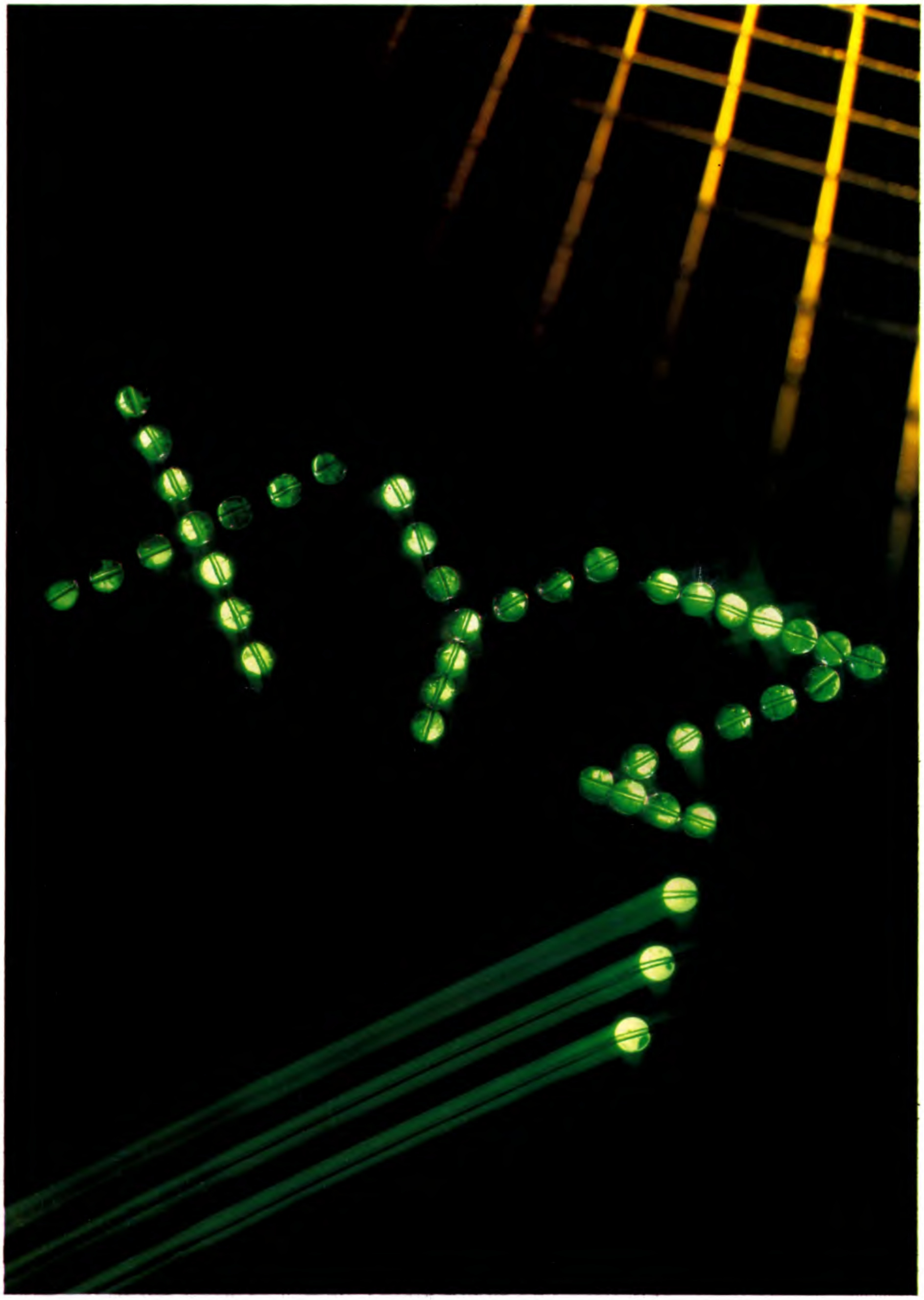
The address of the screen varies, depending on which MODE you are in. The end address for every MODE is 8ØØØ hexadecimal. The start address of each MODE is: Ø—3ØØØ; 1—3ØØØ; 2—3ØØØ; 3—4ØØØ; 4—58ØØ; 5—58ØØ; 6—6ØØØ; 7CØØ.

To LOAD the screen picture back, type:

```
*LOAD filename
```

You can use the display as a title page to a game by LOADing it in first so it stays on the screen while the game is loading. First create a short loader program:

```
1Ø VDU 21
2Ø *LOAD filename
3Ø CHAIN "GAME"
```



The VDU 21 turns off output to the screen so the loading messages don't spoil your display. This does mean that the first line of your main program should be VDU 6 to reset the effect of the VDU 21. The three programs should be positioned on tape in the correct order—LOADER, SCREEN then GAME.

To SAVE a screen to tape from your Dragon or Tandy, you can use this command:

```
CSAVEM"filename",1536,7679,35725
```

The first two numbers are the start address and end address of the screen memory (the first page in any PMODE, unless you have a disk drive. Except when you are using several graphics screens, these are the addresses that you should use). CLOADM will LOAD the picture back into memory again.

You can use the display as the title page to a game by adding a loading routine at the start. Add these lines to the start of your program:

```
1 PMODE1,1: PCLS: SCREEN1,Ø
2 CLOADM
3 FOR D=1 TO 1ØØØ: NEXT
```

and make sure that the screen picture is SAVEd immediately after your game. However, this only works as long as the game doesn't reduce the number of graphics pages available from start-up, or the screen picture may overwrite part of your BASIC program.

Although the Commodore computers cannot easily LOAD in a screen picture in the same way as the other computers can, they can still SAVE large letters to tape. If you do not wish to use the actual programs to create the large letters as a subroutine in your own program, you can SAVE the routine as a block of memory to tape, and alter the character set pointer so that it points to your block of memory, and you can then PRINT the large letters in a series of PRINT statements.

The three types of large letters you have seen in this article, and the one in the last issue, are not the only kind of large letter you can create with your computer. You have already seen that one way you can store your large letters is to put them into UDGs. You could equally well design several UDGs together, which combine to form a large letter. Using this method, you can design as intricate UDGs as you like, since the computer has only to PRINT a string—it does not have to actually draw each character, as the other programs do.

ADDING INSTRUCTIONS TO BASIC

BASIC is just another machine code program that runs on your computer. And even though the instructions that are executed by the processor are fed to it from ROM—instead of from the keyboard or from tape or disk—on most micros it is possible to tamper with BASIC, even to the extent of adding, or deleting, commands.

Adding instructions can be useful if you are using your computer in a very specific application that needs complex routines in BASIC over and over again. In this article a couple of instructions each are added to the Spectrum's BASIC (if you have Interface 1 and 48K), and the BASIC's of the Acorn and Dragon. Commodore 64 users have already been putting the theory to practical use by adding the graphics instructions that are missing from Commodore's BASIC.

Adding an instruction to BASIC has to be done in two operations. First, a routine has to be added to the BASIC editor so that it recognizes the syntax of the new instruction, otherwise it will kick up an error message. Then a routine has to be added to the main body of the BASIC to execute the new instruction when the program is RUN.

This next routine will add INVERSE and ATTR instructions. Note that these keywords already exist on the Spectrum. But normally INVERSE has to be followed by Ø or 1. Ø leaves everything as normal, and 1 INVERSEs all output to the screen from that point onwards. The new INVERSE being added here takes no parameters and INVERSEs the whole screen.

The Spectrum's ATTR is normally a function, not an instruction. You use it to ascertain the ATTRibute of a screen location specified by two parameters in brackets which follow the keyword. The new ATTR is a command. It takes one parameter (not in brackets) which specifies the ATTRibute of the whole screen.

Remember, this routine only works if you have Interface 1 attached.

```
10 REM org 65200
20 REM rst 16
30 REM defw $0018
40 REM cp 221
50 REM jr z,inv
60 REM cp 171
70 REM jr z,attr
80 REM jp $01F0
90 REM inv rst 16
100 REM defw $0020
110 REM call $05B7
120 REM ld hl,$4000
130 REM ld b,24
140 REM invlo push bc
150 REM ld b,0
160 REM invli ld a,(hl)
170 REM cpl
180 REM ld (hl),a
190 REM inc hl
200 REM djnz invli
210 REM pop bc
220 REM djnz invlo
230 REM jp $05C1
240 REM attr rst 16
250 REM defw $0020
260 REM rst 16
270 REM defw $1C82
280 REM call $05B7
290 REM rst 16
300 REM defw $1E94
310 REM ld hl,22528
320 REM ld (hl),a
330 REM ld de,22529
340 REM ld bc,767
350 REM ldir
360 REM rrca
370 REM rrca
380 REM rrca
390 REM out 254,a
400 REM jp $05C1
410 REM end
900 CLEAR 65199
```

The origin of this routine is 652ØØ. But it is not called in the usual way. You do not want it running all the time, otherwise the routine that runs BASIC would not be able to operate. Instead, it is wedged in between BASIC and the error messages.

When the BASIC editor has run and finds that it does not recognize the instruction or syntax in the edit line, it looks at the VECTOR system variable. This usually directs it to the error message routines at Ø1FØ. But if the start address of another routine, like this one,

The BASIC on your computer is not something that is fixed. It is a program, like any other, and you can customize it by adding your own instructions

- WEDGING NEW INSTRUCTIONS
- CHANGING FROM ROM TO ROM
- REDIRECTING VECTORS
- NEW WORD TABLES
- CHECKING SYNTAX
- WHEN WORDS FAIL
- ADDING NEW STUBS
- RECOGNISING NEW TOKENS
- **OLD**ING YOUR DRAGON
- SWITCHING COLOUR SETS

is POKEd into VECTOR the processor will go off and execute that routine instead.

SWITCHING BETWEEN ROMS

Interface 1's own built-in ROM itself adds new instructions to BASIC. These are mainly concerned with Microdrive and handle its SAVE and LOAD instructions. But it does add other instructions like CLS # which clears the screen back to the power-up colours.

Adding Interface 1 does make it necessary to switch back and forth between the new ROM in the interface and the old ROM in the Spectrum, though. Normally, the processor looks at the new ROM in Interface 1. But **rst 16**—that is, **rest**art **16**—sends it to the new ROM's restart routine at ØØ1Ø which directs it on to the old ROM. But the restart routine does not just send the processor to any old place in the old ROM. It sends it specifically to the routine which starts at the address given by the two bytes that follow the **rst** instruction.

So **rst 16, defw $ØØ18** jumps to the routine at ØØ18 in the old ROM. This routine does what **rst 24** would do if Interface 1 had not been plugged in. It gets the command code which occurs at the beginning of the next BASIC statement.

The **cp 221** compares the command code with the INVERSE token. And if they match, **jr z, inv** sends the processor to the **inv**erse routine.

If they do not match, **cp 171** compares the command code with the token for ATTR. If this matches, **jr z,attr** sends the processor off to execute the **attr**ibute routine. And if the token for **attr** does not match either, **jp$Ø1FØ** sends the processor to the syntax error routine at 496, which is where it would have gone in the first place if this routine had not intercepted it.

THE INVERSE ROUTINE

The **rst 16** switches to the old ROM again and calls the routine at ØØ2Ø which moves the pointer onto the next byte of the line of BASIC.

The routine at Ø5B7 is then **call**ed. This checks for an end of statement marker. If it finds one in syntax time—that is, when the BASIC editor is working on a line prefixed with a line number—it exits ready to accept the next line or a direct command.

If the computer is in running time—that is, a direct command is being entered or a program is being RUN—this routine sends the processor back into this routine to execute the next instruction.

The **ld hl,$42ØØ** loads the hl register with the first screen address. The B register is then set up as a double counter. First, the number 24 is loaded into B and pushed onto the stack, then Ø (which will act as 256 when you start decrementing it) is loaded. There are 24 × 256 screen locations.

The instruction **ld a,(hl)** loads the contents of the first screen location—that is, the top lefthand corner—into the accumulator. The **cpl** **compl**ements it. It turns all the Ø bits to 1s and all the 1s to Øs. Then **ld (hl),a** puts the outcome of this operation back into the same screen location.

The result is to turn on all the pixels that were off, and all the pixels that were on, off. In other words, it inverses that part of the screen.

COUNTING ACROSS THE SCREEN

HL is then incremented to move onto the next screen location. The **djnz invli** decrements the B register and jumps back to **invli** if B hasn't counted down to zero. So this loop is executed 256 times, each time moving onto the next location and inversing it. When it has counted down to Ø the other counter is popped off the stack, decremented, and if it is not zero, the processor jumps back to **invlo** where the counter is pushed back onto the stack.

This outer loop is executed 24 times, and the inner loop is executed 256 times each time the processor goes round the outer loop—making 6,144 times in all. When the counter for the outer loop has counted down to zero and every screen location has been inversed, **jp $Ø5C1** takes the processor to the routine at Ø5C1 which directs it to the next BASIC statement so that it can start executing it.

THE ATTRIBUTE ROUTINE

This routine begins in exactly the same way as the INVERSE routine—it sends the processor to the routine in the old ROM which moves the pointer onto the next byte after the keyword.

On this occasion, though, this should be a parameter. So **rst 16** and **defw $1C82** send the processor to the routine in the old ROM which evaluates numerical expressions. It puts the result on the stack.

The routine at Ø5B7 is called again. This is the one that checks for the end of a statement and exits in syntax time. If there is no end-of-statement marker after the numerical expression, but a token, a letter, a punctuation mark or another numerical expression, the routine will give an error message. The new command takes only one parameter, so if there is anything else following it the syntax is wrong.

The Spectrum will not get confused if you use the normal ATTR function which takes two parameters. That will already have been picked up earlier by the Spectrum's own BASIC routines. It would have been approved and entered, and the processor would not be on its way to the error message routines where this program intercepted it. So both the old INVERSE function and the new INVERSE command and the two ATTRs will be accepted.

The **rst 16** and **defw $1594** send the processor to the routine in the old ROM that picks the value of the parameter back off the stack. It puts it into the accumulator.

The **ld hl,22528** then loads the address of the beginning of the attribute file into HL. The **ld (hl),a** loads the numerical value of the INVERSE parameter into the first location in the attributes file.

The **ld bc,767** loads the BC register with 767. It is going to be used as a counter. There are 768 locations in the attributes file—24 rows by 32 columns—but one of them has already been dealt with. The **ldir** block-loads the contents of the location pointed to by HL into the location pointed to by DE, increments HL and DE, decrements BC and repeats if the result is not zero.

In other words, it copies the attribute of the first screen location—the one the program has just set—into the attributes of all the other screen locations.

The next thing to do is to change the border colour to match the paper colour on the screen. To do this the **out** command is used. The problem is that the paper colour is stored in bits 3, 4 and 5 of the attributes, while the border colour has to be in bits Ø, 1 and 2 with the **out** command.

So the three **rrca**s simply shove the attribute value, which is still in the accumulator, three places to the right. It is then **out**ed through port 254.

Once all that is done **jp $Ø5C1** takes the processor to the routine at Ø5C1 again.

RUNNING THE PROGRAM

The CLEAR command at the end of the program automatically moves the top of BASIC to protect the machine code. To get this program to run each time one of the new commands is entered, VECTOR must contain the start address of this routine.

So you have to put it into the two-byte pointer with the following POKEs:

```
POKE 23735,176:POKE 23736,254
```

Be very careful when keying these POKEs. If you get either of them wrong and then key in a syntax error, the processor will be directed to the wrong place and would crash.

It is even more critical that you get the second POKE statement correct. The first POKE changes half the VECTOR. So if there is a syntax error in the second half the processor will be misdirected disastrously.

The following assembly language program adds the instructions INV and BYE to the BBC's BASIC. The BBC's OS offers a facility which allows you to add new commands easily. But they must be prefixed by *LINE, followed by a space.

```
10 MC=&AØØ
20 JTABLE=&9ØØ
30 WORD=JTABLE+&4Ø
40 FOR T=Ø TO 3 STEP 3
50 P%=MC
60 [OPT T
70 STX &72
80 STX PT2+1
90 STY &73
100 STY PT2+2
```

```
110 LDA #WORD□MOD 256
120 STA &7Ø
130 LDA #WORD□DIV 256
140 STA &71
150 LDX #Ø
160 LDY #Ø
170 STY &74
180 .PT: LDA (&7Ø),Y
190 PHA
200 AND #127
210 .PT2:CMP &FFFF,X
220 BNE FAIL
230 INX
240 INY
250 PLA
270 BPL PT
280 LDA &74
290 ASL A
300 TAY
310 LDA JTABLE,Y
320 STA JUMP+1
330 LDA JTABLE+1,Y
340 STA JUMP+2
350 .JUMP:JMP &FFFF
360 .FAIL:INC &74
370 PLA
380 LDX #Ø
390 .F2:LDA (&7Ø),Y
400 INY
410 AND #128
420 BPL F2
430 LDA (&7Ø),Y
440 BPL PT
450 BRK
460 ]
470 $P%=CHR$(255)+"Foul up"
    +CHR$(Ø)
480 P%=P%+9:[OPT T
490 .FIRST:LDA #17:JSR &FFEE:LDA
    #Ø:JSR &FFEE:LDA #17:JSR &FFEE:LDA
    #135:JSR &FFEE
500 TXA:TAY
510 .F2:LDA (&72),Y:INY:JSR &FFEE:CMP
    #&ØD:BNE F2
520 LDA #17:JSR &FFEE:LDA #1:JSR
    &FFEE:LDA #17:JSR &FFEE:LDA
    #128:JSR &FFEE
530 LDA #10:JSR &FFEE
540 RTS
550 .TWO:LDA #135:STA &7Ø:LDA
    #22:JSR &FFEE:LDA #2:JSR &FFEE:LDY
    #30
560 .T2:LDX #20
570 .T3:LDA #17:JSR &FFEE:LDA &7Ø:JSR
    &FFEE:DEC &7Ø:BMI T4:LDA #143:STA
    &7Ø
580 .T4:LDA #32:JSR &FFEE:DEX:BNE
    T3:DEY:BNE T2:RTS
2000 ]:NEXT
2010 ?&200=MC□MOD
    256:?&201=MC□DIV 256
2020 Q=-1
2030 FOR P=Ø TO 1
2040 READ A$,A
2050 ?(JTABLE+P*2)=A□MOD 256
2060 ?(JTABLE+P*2+1)=A□
    DIV 256
2070 FOR T=1 TO LEN (A$)
2080 Q=Q+1
2090 ?(WORD+Q)=ASC
    (MID$(A$,T,1))
2100 NEXT
2110 ?(WORD+Q)=?
    (WORD+Q)+128
2120 NEXT
2130 Q=Q+1
2140 ?(WORD+Q)=128
2150 DATA "INV",FIRST,"BYE",TWO
```

SETTING UP

Line 1Ø gives the start address of the machine code. And Lines 2Ø and 3Ø give start addresses for the jump table and the details of the new BASIC words being added. These should be changed if the cassette buffers are used—these are not used during SAVEing and LOADing, only during file handling.

Then the assembler is initialized.

When the BBC's OS hits a *LINE it puts the high and low bytes of its start address of the next word in the Y and X registers.

In Lines 7Ø and 8Ø the contents of the X register are copied into the zero page 72—where it can be referred to later—and into the first byte past the label PT2. This occurs in Line 21Ø.

The contents of the Y register are stored in the subsequent bytes in Lines 9Ø and 1ØØ. The whole of the start address of the command word following *LINE has now been POKEd into the CMP instruction in Line 21Ø.

LDA #WORD MOD 256 and STA &7Ø then stores the low byte of the start address of the word table. Lines 13Ø and 14Ø store the high byte in &71.

Lines 15Ø and 16Ø then clear the X and Y register and Line 17Ø uses the zero value in the Y register to clear another zero page memory location to store the word number.

COMPARING WORDS

LDA (&7Ø),Y loads the accumulator with the contents of the memory location pointed to by 7Ø and 71, offset by Y. This is the first byte of the first word in the table when the processor first goes round the loop.

PHA pushes it on the stack and AND #127 clears the high bit. The last letter of each word in the word table is marked by having its high bit set—as all the ASCII values of letters of the alphabet are less than 127, it does not affect. So ANDing with 127 will leave a regular ASCII alone but resets the high bit of an end-of-word letter, turning it back into normal ASCII form. Obviously, this ANDing makes no difference on the first pass. (One-letter command words are not allowed.)

The byte in the accumulator is then compared with the letter pointed to by the address following the CMP instruction, offset by X. Because an offset is needed with this instruction, an absolute base address has to be given. You cannot post-index with the X register in an indirect addressing mode.

If the two letters do not match, Line 22Ø sends the processor off to the FAIL routine. But if they do match, Lines 23Ø and 24Ø increment X and Y to move onto the next byte of the word in BASIC and the next byte of the word you're comparing it to in the word table.

The last byte is then pulled off the stack again by PLA, and BPL—Branch if PLus—takes the processor back to deal with the next byte if bit 7 is not set. If bit 7 is set—as it would be if the contents of the register were negative or the routine had matched the last letter of a word in the word table—the processor goes onto the next instruction.

CONSULTING THE JUMP TABLE

When a word has been found and matches a word in the word table completely, the LDA &74 in Line 28Ø loads the word number from the memory location set aside for it. ASL—Arithmetic Shift Left—then doubles it. The ASL A instruction shifts all the bits in the accumulator one bit to the left. And as they are binary bits, this effectively doubles the value of the register in the same way as multiplying a decimal number by ten is done by shifting all the digits one place to the left.

The jump table is made up of two byte addresses, so to count along the address table you need to move two bytes at a time.

TAY puts the result of the doubling back into the Y register so that it can be used as an offset in Lines 31Ø and 33Ø. And Lines 32Ø and 34Ø POKE the values picked up from the jump table into the JMP command in Line 35Ø. The processor then jumps to the routine that's been written to cope with the new command.

THE FAIL ROUTINE

If one of the letters in the command word in BASIC does not match the corresponding letter in the word in the word table, the processor is sent to the FAIL routine on Line 36Ø. The first thing it does is to increment the word counter in memory location 74, so it is ready to check whether the next word in the table matches.

The PLA in Line 37Ø pulls the last item off the stack. This is not going to be used, but it must be pulled off, otherwise the stack would grow uncontrollably.

Then the X register is reset to zero, ready to count along from the first letter of the word in BASIC again.

The next byte of the word in the word table is loaded up into A and the offset Y is incremented. Then the contents of A are ANDed with 128, and BPL checks for a positive result. The AND #128 is necessary this time because the INY sets the sign flag too, and it comes after the LDA.

The F2 loop is executed over and over until a byte with bit 7 set is found. The loop is used to clock up the Y counter until it points to the first letter of the next word.

At the end of the word table there is a 128. And the instructions in Lines 430 and 440 check for that. If the 128 is not found, the processor jumps back to the beginning of the comparing routine again.

If not, it goes onto the BReaK command BRK. The processor then returns to BASIC and Line 470 prints the error message 'Foul up' on the screen.

THE INV COMMAND

The INV command that is being added prints whatever follows the INV in black on a white background, instead of the other way round.

Line 480 puts the BBC's assembler back into operation. Following the label FIRST, the instructions LDA #17:JSR &FFEE are equivalent to COLOUR. And when they are followed by LDA #0:JSR &FFEE, the effect is the same as COLOUR 0 in BASIC, which gives black letters.

The assembly language equivalent of COLOUR is repeated, this time followed by 135 which gives a white background.

Line 500 transfers the contents of the X register into the Y register. This can only be done via the accumulator. If you look back, you will see that the X register carries the position directly after the command INV. This needs to be in the Y register because you can't do post-indexed indirect addressing with the X register.

In Line 510, LDA (&72),Y picks up the first letter of the word following the INV command. Then Y is incremented so that it points to the next location, while JSR &FFEE outputs the first letter, reversed out, to the screen. CMP #&0D checks for a carriage return. And if there isn't one, BNE F2 takes the processor back again to print out the next letter.

Once a carriage return has been located, Line 520 gives the assembly language equivalent of COLOUR 7 and COLOUR 128, which turns the screen back to white letters on a black background. Then LDA #10:JSR &FFEE gives a line feed and RTS returns to BASIC.

THE BYE COMMAND

BBC BASIC is so comprehensive that it is difficult to come up with sensible commands to add that have any general application. So as a demonstration the BYE command simply gives a checkered pattern on the screen.

The number 135 is stored in location 70. The instructions LDA #22:JSR &FFEE is the assembly language equivalent of MODE, so the next two instructions switch to MODE 2. Then Y is loaded with 30 and X is loaded with 20—the screen is 30 rows by 20 columns.

LDA #17:JSR &FFEE gives COLOUR again. And the colour parameter is loaded from location 70. The number in that location is then decremented and BMI checks where it has gone positive—in other words, it has been decremented below 128. If it has, 143 is stored in 70. If not, these instructions are jumped and the processor goes straight onto LDA #32:JSR &FFEE which prints a space in the background colour on the screen. X is then decremented to move onto the next character square and the print process is repeated with a colour parameter one lower. When X has been decremented to 20, Y is decremented and the whole process starts again until the whole screen is filled.

RUNNING THE PROGRAM

To assemble the routines, the program is RUN in the normal way. But it is not called like a regular machine code program. Instead, the vectors in 200 and 201 point to routines that need to be called after *LINE. Line 2010 pokes the start address into these vectors.

Line 2050 and 2060 poke the low and high bytes of the command routines into the jump table. Line 2090 creates the word 'table'. Line 2110 sets the high byte of the last letter of a word and Line 2140 puts the 128 end-of-table marker in. Up to 32 words can be added as long as the total is less than 192 characters.

The following routine adds two new commands to the Dragon's BASIC. These are OLD, which allows you to revive a program if you have mistakenly NEWed it—and INVERT, which swaps the two colour sets.

```
        ORG    31000
SETUP   LDX    #298
        LDU    #308
STONE   LDA    ,X+
        STA    ,U+
        CMPX   #308
        BLO    STONE
        LDA    #2
        STA    298
        LDX    #NEWRDS
        STX    299
        LDX    #NEWDSP
        STX    301
        LDX    #NEWUSR
        STX    176
        LDU    #$8B8D
        LDA    #10
STTWO   STU    ,X++
        DECA
        BNE    STTWO
        RTS
NEWRDS  FCB    79,76,196
        FCB    73,78,86,69,82,212
NEWDSP  CMPA   #$CE
        BLO    NDONE
        CMPA   #$D0
        BHS    NDONE
        SUBA   #$CE
        LDX    #NEWTBL
        JMP    $84ED
NDONE   JMP    $89B4
OLD     LDU    25
        PSHS   U
        LEAX   4,U
OLDONE  LDA    ,X+
        BNE    OLDONE
        TFR    X,D
        SUBD   ,S++
        TSTA
        BEQ    OLDTWO
        JMP    $8B8D
OLDTWO  STX    ,U
OLDTHR  TFR    X,U
        LDX    ,U
        BNE    OLDTHR
        LEAU   2,U
        STU    27
        STU    29
        STU    31
        RTS
INVERT  LDA    65314
        EORA   #8
        STA    65314
        RTS
NEWTBL  FDB    $7964
        FDB    $7989
NEWUSR  EQU    *
```

STUBS

To add new words to BASIC, you have to reserve new command words. And to do this you have to extend the table of reserved words and direct BASIC to the new words.

BASIC commands are pointed to by the command interpretation vectors in the areas of RAM known as *stubs*. There are normally two stubs, each of which occupies ten bytes of

RAM. The second is a dummy, though, which simply acts as an end marker.

When you are adding new instructions, you have to create a new stub which points to your new command routines. But first you must move the second, end-marker stub down memory to make room for it.

The start address of the second stub is 298 and the first six instructions in this program shift it so that it starts at 3Ø8. The X register is used as pointer to the source stub while the U register points to its destination. The ,X+ and ,U+ increment the pointers each time the STONE loop is executed.

Warning: do not use this program with a disk drive. Disk commands use the same stub.

ADDING A NEW STUB

Once the second stub has been moved and the processor has come out of the STONE loop, the new stub has to be created.

The first byte of the new stub should carry the number of commands it is pointing to. So LDA #2 and STA 298 store the number two in the first byte.

The second two bytes carry the address of the table of new command words. These appear after the label NEWRDS with their letters carried byte-by-byte in ASCII. You'll note, though, that last letter in each word is marked by setting the most significant bit to 1. So D—the last letter of OLD—is represented by its ASCII, 68, plus 1ØØØØØØØ in binary or 128 in decimal. (68+128=196.) Likewise, the T of INVERT is 212 instead of 84.

The fourth and fifth bytes of the stub carry the start addresses of the command routines. These are given in the double bytes after the label NEWTBL.

USR VECTOR

The vector at 176 and 177 points to the locations which store the address of the USR routines. These are usually kept in a table starting at 3Ø8, after the end of the second stub. But you have already shifted that stub up into the USR table to make room for the new commands. So the USR vector has to be redirected.

LDX #NEWUSR and STX 176 store the address of the label NEWUSR in the 176 and 177. And the EQU* after NEWUSR at the end of the program simply reserves the following bytes for the USR table.

THE NEW TOKENS

The highest token usually used in Dragon BASIC is CD. So the two new tokens for your two new instructions are CE and CF. CMPA #$CE and BLO NDONE check where the token is below the range of the new tokens. If it is, and hasn't been picked up by the first stub, there must be something wrong with the syntax. So the processor is sent to the label NDONE which, in turn, jumps to 89B4— the syntax error message routine in ROM.

CMPA #$DØ and BHS NDONE makes the same Branch if the token is Higher than, or the Same as, DØ. So this sends the processor off to the ROM error message routine if the token is out of the other end of the range.

If the token has passed those two tests, it is inside the range of the new tokens you're defining and the processor continues to the next instruction. The value of the token in A then has CE subtracted from it. The result is the number of the new token. It is left in A.

The X register is loaded with the start of the new command routine's address table. So when the processor jumps to 84ED—the routine that dispatches the processor to the various BASIC command word routines—it carries with it the parameter in X and A.

THE OLD ROUTINE

When a BASIC program is NEWed, it resets several pointers. So if these pointers are reset before the program has been overwritten or the BASIC area has been moved by a PCLEAR it is possible to salvage the old program.

The pointers effected by NEWing are the one in the first two bytes of the first line of BASIC, which usually points to the beginning of the next line, and the system variables that point to the end of the BASIC program area. The OLD command simply re-instates these pointers with their former values.

The OLD routine begins with the instruction LDU 25. This loads the U register with the contents of memory locations 25 and 26, which are the pointer to the beginning of

BASIC. PSHS U pushes this onto the stack so that it can be recalled for a subtraction later in one instruction.

LEAX 4,U loads the X register with the address of the byte four on from the start of BASIC. The first two bytes of any line of BASIC, remember, contains the address of the start of the next line. And the next two bytes contain the BASIC line number. So this instruction loads up the address of the first byte of the actual line.

LDA ,X+ loads that byte into the accumulator and increments X. BNE OLDONE sends the processor back to perform this instruction over and again until it hits a zero, that marks the end of the BASIC line.

When it finds one and breaks out of the loop, X has already been incremented so that it points to the beginning of the next line of BASIC.

TESTING FOR CORRUPTION

If the OLD command is invoked when there is no recognizable program left in BASIC, you don't want it to return a lot of gobbledegook. So a test that checks whether the first line still makes some sort of sense is required.

To do this, the pointer in X is transferred into D, then SUBD ,S++ subtracts the last item on the stack—the address of the beginning of BASIC—from the contents of D which is the address of the start of the second line of BASIC. The result is the length of the first line and is left in D. And the stack pointer is decremented, effectively pulling the address of the beginning BASIC off the stack.

TSTA TeSTs A—that is, it looks at the contents of the high byte of the D register (the A and B registers together make up D) and sets the flags accordingly. BEQ OLDTWO then jumps over the next instruction if the zero flag is set. But if the contents of A are not zero, so the contents of D are more than 255—the maximum number of bytes that can be occupied by a line of BASIC. In that case the branch is not made and the processor jumps to the error message routine at 8B8D.

RE-INSTATING THE POINTERS

The U register still points to the first byte of the first line of BASIC. So storing the value of X in the address pointed to by U, with STX ,U re-instates the 'beginning of next line' pointer.

The next thing that you have to do is work out where the end of BASIC is. This is done simply with the OLDTHR loop.

As the two bytes at the beginning of each line point to the beginning of the next line, it is simple to jump up from line to line by transferring the pointer in X into U and loading X with the contents of the location pointed to by U. X contains the address of the beginning of the next line. That is put into U. U then points to the two bytes that contain the address of the line of BASIC after that. And this is loaded into X.

This loop is executed over and over again until it hits a line that starts with two bytes containing ØØ ØØ. This is the end of the BASIC program. And when it is found the BNE OLDTHR no longer branches.

U is then loaded with the address of the location two on from the beginning of this marker. This simply jumps the pointer in U over the two zero bytes so that it points to the next one—which should be the beginning of the variables area.

STU 27, STU 29 and STU 31 then copies this address into the system variable at 27 and 28, which is the pointer to the start of the variables area, 29 and 3Ø, which is the pointer to the start of the array table, and 31 and 32, which points to the end of RAM in use.

RTS then returns to BASIC.

THE INVERT ROUTINE

The memory location FF22 in the input/output area of memory controls the graphics output. And bit 3 of that location controls the colour set being used.

LDA $FF22 loads the contents of that location into the accumulator and bit 3 is flipped by exclusively oring it with 8—which is ØØØØ1ØØØ in binary. The result is then stored back in FF22 and RTS returns to BASIC.

This routine simply changes between the two colour sets, flipping everything that is one into the other.

This program is not relocatable.

FINDING A WAY WITH WORDS

If you want to tidy up your typing without the expense of a fully-fledged wordprocessor, this easily-programmed text editor is a simple answer

The virtues of wordprocessing have already been extolled in a previous issue of *INPUT* (pages 541–543), and it is, indeed, one of the most useful applications for any home computer. To get the best results, you can buy top-flight machine code software, but this is usually designed for professional use and can prove costly.

The part of the program covered by this article represents a good standard first step text editor which enables any first-time user to get a real 'feel' for a program of this type.

With this program you can create memos, letters or any other form of correspondence for output to a printer. Text can be saved to tape or disk (or Microdrive on the Spectrum) and then recalled for further use by this program or any other which can make use of the sequential data files created.

Because of the program's overall length—at least 7.5K—the listing is split up into three easily digestible units and *will not RUN properly until the last part has been entered.* However, you will be able to create text files after entering the second part. The third part consists of the printer routine which you'll need to obtain printouts of your text files. Sort and search facilities plus a form letter option are also included.

The length of the program also makes it unsuitable for the unexpanded Vic 20. Because of this, and because the Vic's screen will only display a narrow measure of text, there is no Vic version.

Before you enter the first part, which follows, let's look at how the program is used. It is set up for convenient use in most home applications, but as with any program of this type, you can customize the various menus and screen displays to your heart's content. If you do decide to adapt it, however, do stick to the same line numbering conventions so that the program links correctly, particularly when further modules are added.

THE MAIN MENU

When you RUN your completed copy of the program, you are immediately prompted for choice of input/output options. This sets the system up for tape and/or disk use, which you can choose by pressing either T or D. Spectrum users who have a Microdrive can adapt the program for this by making the indicated program amendments.

Note that you can input from one device and output to another, and change devices later should you wish to do so.

The general menu is then displayed and you are presented with the following options:

[L]OAD
[S]AVE
[I]/O CHANGE (except on Spectrum)
[E]DIT
[C]LEAR
[P]RINT
[Q]UIT

LOAD

If you wish to continue working on existing text which has previously been SAVEd, press L to initiate the LOAD routine. On the Spectrum the options are numbered. As any text presently in memory is overwritten in the process, an 'Are You Sure' message is displayed. Press Y to continue (or any other key to abort). You are asked for a file name and have to make an entry to proceed. The nominated text file then LOADs into memory and can be worked on as required.

SAVE

The SAVE routine is used for creating sequential data storage files. When you select this option by pressing S—and providing there's something in memory—you are first asked for a file name. The Spectrum saves irrespective of whether you have any text in memory. A null entry won't be accepted. The system already 'knows' which device you are using and continues to prompt you accordingly on preparing the tape or disk unit. Finally, press RETURN or ENTER to effect the SAVE.

Dragon and BBC disk users must be careful over the choice of names and procedures if named files are not to be overwritten unintentionally. Commodore users should resist using the "@0:filename" disk SAVE procedure except when the text file to be overwritten is the last one listed.

Additional file SAVE routines can be added to permit file protection, renaming, and replacement. Specimen routines are given in the listing in the next part of this article.

INPUT/OUTPUT

If, for any reason, you decide to change the start-up input/output parameters, you can do so at any time by selecting I from the main menu. Key in your choice. This automatically sets up the program for the new configuration. The Spectrum program is for tape *or* Microdrive. The appropriate amendments for the latter appear after the main listing.

EDIT MODE

Selecting E puts you into edit entry mode and a secondary menu is displayed:

- SCANNING THE MENU
- LOADING AND SAVING
- FILE NAMES
- EDITING
- ADJUSTING YOUR COLOURS
- INPUT/OUTPUT PARAMETERS
- MANIPULATING TEXT
- CLEARING MEMORY
- TEXT EDITOR PROGRAM

[T]OP
[B]OTTOM
[N]EXT LINE
[C]OLOUR (not Dragon/Tandy or
[M]ENU Commodore)

You can select one of the first two options to take you to the start (TOP) or end (BOTTOM) of the text you have in memory. Or, using the third option, you can return to the last line you were working on. Or you can exit to the main menu.

Using the COLOUR option you can adjust the screen 'ink', 'paper' and 'border' colours to your liking.

Each of the first three options puts you into entry mode. The screen displays lines depicting the top and bottom of copy (it may display only one or the other if there's already text in memory). Towards the lower part of the screen there's a separate 'work area' and a 'memory free' indicator (not on the Spectrum) which lets you know how many unused characters remain.

Text entered in the lower area is transferred to memory and to the display area when RETURN or ENTER is pressed, or automatically when two screen lines have been completed.

A number of editing features are built into the program. These (and the keys which control them) vary slightly between machines and are discussed in greater detail in the relevant sections. Note that all editing and entries have to take place in the work area.

Editing controls enable you to move back and forwards along the line of text in the work area for insertion or deletion of characters *before* the text is transferred to memory.

Text already in memory—in other words, displayed in the upper panel—has to be *copied* a line at a time to the work area for amendment.

New lines can be entered at any point in the main body of text. Do this by pressing the appropriate key for 'editor' mode (see the detailed instructions) and position the marker *below* the point where you wish to insert the new line.

Text can likewise be removed a line at a time by entering editor mode and positioning the marker again below the victim line before pressing the appropriate delete key.

Control keys are provided to enable you to jump forwards or backwards through text in memory ten lines at a time, except on the Spectrum. This facility can be used to preview text prior to printing or editing.

You can return to the editor mode menu by pressing the designated 'escape' key, and from there to the main menu for resetting any of the system parameters.

CLEAR OUT

A further option available from the main menu is the clear memory function—but only after you've answered 'yes' to the 'Are You Sure?' prompt which follows keypress C. But this option doesn't reset the input/output or printer parameters. If you wish to start right from scratch, pressing Q to quit does a complete system reset and returns the computer to BASIC.

```
10 POKE 23659,3
20 BORDER 7: PAPER 7: INK 9: CLS
30 DIM l(8): FOR n=1 TO 8: READ l(n):
   NEXT n
40 LET Z$=“ ”
100 LET ext=200: DIM t$(ext,32): LET
   ll=32: LET pl=32
105 LET s$=“□□□□□□□□□□
```

```
   □□□□□□□□□□
   □□□□□□□□□□□
   □□□□"
110 LET t$(1)="TOP OF TEXT
      FILE": LET t$(2)="------
   -------------
   -------------
   ---": LET t$(3)=s$
120 LET t$(4)=s$: LET t$(5)="---
   -------------
   -------------
   ------": LET t$(6)="END OF
   TEXT FILE"
130 LET t=1: LET b=6: LET p=4
140 CLS :PRINT INVERSE 1;AT 0,10;"□MAIN
   MENU□"
150 PRINT AT 4,8;"1:— Load text";AT
   6,8;"2:— Save text";AT 8,8;"3:— Change
   paper";AT 10,8;"4:— Enter editor";AT
   12,8;"5:— Clear text";AT 14,8;"6:— Print
   out text";AT 16,8;"7:— Alter printer";AT
   18,8;"8:— Quit program"
160 PRINT #1;TAB 7;"Select option (1–8)"
170 LET a$=INKEY$: IF a$="" THEN GOTO
   170
180 IF a$<"1" OR a$>"8" THEN GOTO
   170
190 LET a=VAL a$: CLS
200 GOSUB l(a)
210 GOTO 140
500 CLS : PRINT INVERSE 1;AT 4,10;
   "□EDITOR MENU□"
510 PRINT AT 8,8;"1:— Top of text";AT
   10,8;"2:— End of text";AT 12,8;"3:— Next
   line of text";AT 14,8;"4:— Quit edit menu"
520 PRINT AT 18,7;"Select option (1–4)"
530 LET a$=INKEY$: IF a$="" THEN GOTO
   530
540 IF a$<"1" OR a$>"4" THEN GOTO
   530
550 LET a=VAL a$: CLS
560 IF a=4 THEN RETURN
570 IF a=1 THEN LET p=4
580 IF a=2 THEN LET p=b-2
590 GOSUB 1000: GOSUB 2000
600 GOTO 500
900 PRINT AT 10,8;"Are you sure?": PAUSE 0
910 IF INKEY$="y" THEN RUN
920 RETURN
4000 RETURN
6000 REM load
6010 INPUT "Enter filename", LINE n$: LOAD
   n$ DATA t$()
6020 LET b=VAL t$(1): LET t$(1)="TOP OF
   TEXT FILE": RETURN
6200 REM save
6210 LET t$(1)=STR$ b
6220 INPUT "Enter file name", LINE n$: IF
   n$="" OR LEN n$>10 THEN GOTO
   6220
6230 SAVE n$ DATA t$(): GOTO 6020
6500 INPUT AT 0,0;"Enter printer width
   (1–80)",pl: IF pl<1 OR pl>80 THEN
   GOTO 6500
6510 INPUT AT 0,0;"Enter
   characters per line
(1–";(pl);")";ll: IF ll<1 OR ll>pl THEN
   GOTO 6510
6520 LET ll=ll+1: RETURN
9000 DATA 6000,6200,3000,500,900,
   4000,6500,9999
```

To adapt for Microdrive use, omit Lines 6010, 6020, and 6230 and add the following lines:

```
6005 CAT 1
6010 INPUT "Enter filename", LINE n$: IF LEN
   n$<1 OR LEN n$>10 THEN GOTO 6010
6015 LOAD *"m",1; n$ DATA t$()
6205 CAT 1
6230 SAVE *"m",1; n$ DATA t$(): GOTO
   6020
```

```
10 CLS
20 PMODE0:PCLEAR1:CLEAR17500
30 DIM TX$(500)
40 BL$=CHR$(128):TL=1:CP=1:MW
   =80:TW=60:PL=66:TH=60:GP=10:
   LF$=STRING$(3,13):GOSUB5000
50 TX$(0)=STRING$(32,195)
60 TX$(TL)=STRING$(32,188)
70 CLS:PRINT@10,BL$"main"BL$
   "menu"BL$:PRINT@104,"(L)OAD":
   PRINT@136,"(S)AVE":PRINT@168,
   "(I)/O CHANGE"
80 PRINT@200,"(E)DIT TEXT":PRINT
   @232,"(C)LEAR MEMORY":PRINT
   @264,"(P)RINT OUT":PRINT
   @296,"(A)LTER PRINTER":
   PRINT@328,"(Q)UIT PROGRAM";
90 B$=INKEY$:IF B$="" THEN 90
100 B=INSTR("LSIECPAQ",B$):
   IF B=0 THEN 90
110 ON B GOSUB 4500,4000,5000,
   1000,160,3000,5500,130
120 GOTO 50
130 CLS:PRINT" ARE YOU SURE (Y/N) ?"
140 R$=INKEY$:IF R$<>"Y" AND
   R$<>"N" THEN 140
150 IF R$="Y" THEN CLS:END ELSE RETURN
160 CLS:PRINT@8,BL$;"clear";BL$;
   "memory";BL$:PRINT:PRINT" ARE YOU
```

```
   SURE (Y/N) ?"
170 B$=INKEY$:IF B$<>"N" AND
   B$<>"Y" THEN 17Ø
18Ø IF B$="N" THEN RETURN
19Ø FORK=1 TO TL:TX$(K)="":
   NEXT:TL=1:CP=1:RETURN
1ØØØ CLS:PRINT@42,BL$"edit"BL$
   "menu"BL$:PRINT@1Ø4,"tOP OF
   TEXT":PRINT@168,"bOTTOM OF
   TEXT":PRINT@232,"nEXT LINE OF
   TEXT":PRINT@296,"mAIN MENU"
1Ø1Ø B$=INKEY$:IF B$="" THEN 1Ø1Ø
1Ø2Ø B=INSTR("TBNM",B$):IF B=Ø THEN
   1Ø1Ø
1Ø3Ø ON B GOTO 1Ø5Ø,1Ø6Ø,1Ø7Ø,
   1Ø8Ø
1Ø5Ø CP=1:GOTO 1Ø7Ø
1Ø6Ø CP=TL
1Ø7Ø GOSUB 2Ø9Ø:GOSUB 15ØØ:
   GOTO 1ØØØ
1Ø8Ø RETURN
15ØØ A$="□"
151Ø P=Ø:PRINT@384,A$
152Ø CH=PEEK(14Ø8+P):T$=INKEY$:
   IF T$="" THEN CH=(CH+64)AND
   127:POKE14Ø8+P,CH:CH=(CH+64)
   AND127:POKE14Ø8+P,CH:GOTO152Ø
153Ø IF LEN(A$)=65 OR T$=
   CHR$(13) GOSUB2ØØØ
154Ø IF T$="↑" THENSF=Ø:GOSUB
   25ØØ:GOTO151Ø
155Ø IF P<LEN(A$)-1 AND T$=CHR$
   (1Ø) THENA$=LEFT$(A$,P)+MID$
   (A$,P+2):GOTO16ØØ ELSE IF T$=
   CHR$(1Ø) THEN 152Ø
156Ø IF T$=CHR$(12) THEN RETURN
157Ø IF T$=CHR$(21) THEN P=
   -(LEN(A$)-1)*(P=Ø):GOTO16ØØ
158Ø IF T$<>CHR$(8) AND T$<>
   CHR$(9) AND ASC(T$)<32 THEN 151Ø
159Ø IF T$<>"" AND T$<>CHR$
   (8) AND T$<>CHR$(9) THEN A$=
   LEFT$(A$,P)+T$+MID$(A$,P+1):
   P=P+1
16ØØ PRINT@384,A$
161Ø IF T$=CHR$(9) AND P<LEN
   (A$)-1 THEN P=P+1
162Ø IF T$=CHR$(8) AND P>Ø THEN
   P=P-1
163Ø GOTO 152Ø
2ØØØ X=1:IF LEN(A$)>33 THEN X=2
2Ø1Ø FOR K=TL+X□TO CP+X STEP-1:
   TX$(K)=TX$(K-X):NEXT
2Ø2Ø IF LEN(A$)>33 THENTX$(CP)=
   LEFT$(A$,32):TX$(CP+1)=MID$
   (A$,33,LEN(A$)-33) ELSE TX$(CP)=A$
2Ø3Ø FOR K=Ø TO X-1
2Ø4Ø IF RIGHT$(TX$(CP+K),1)="□"
   THENTX$(CP+K)=LEFT$(TX$(CP+K),
   LEN(TX$(CP+K))-1):GOTO2Ø4Ø
2Ø5Ø NEXT
2Ø6Ø A$="□":P=Ø:PRINT@384,A$
2Ø7Ø PRINT@416,""
2Ø8Ø TL=TL+X:CP=CP+X
2Ø9Ø IF CP<5 THEN ST=Ø ELSE
   ST=CP-5
21ØØ PRINT@Ø,;:FORK=ST TO ST+9:
   PRINT TX$(K);:IF LEN(TX$(K))<
   32 THEN PRINT
211Ø IF K=CP-1 THEN PRINT">"
212Ø NEXT:PRINTSTRING$(32,14Ø)
213Ø PRINT@481,BL$;"mem";BL$;
   "free=";32*(5Ø1-TL);BL$;BL$;
   "clear=menu";BL$;:POKE1534,
   32:POKE1529,61:RETURN
```

```
1Ø *FX4,1
2Ø ON ERROR GOTO 136Ø
3Ø RV$=CHR$(17)+CHR$(Ø)+CHR$(17)
   +CHR$(129):NM$=CHR$(17)+CHR$
   (1)+CHR$(17)+CHR$(128)
4Ø MODE6
5Ø P=Ø:A$=STRING$(12Ø,"□")
6Ø N%=19Ø when using disk drive
7Ø DIM TX$(N%-1)
8Ø TL=1:CP=1:MW=8Ø:TW=6Ø:PL=
   66:TH=6Ø:GOSUB 144Ø:GOSUB 11ØØ
9Ø FOR T=Ø TO N%-1:TX$(T)=
   STRING$(4Ø,CHR$(32)):TX$(T)
   ="":NEXT
1ØØ TX$(Ø)=CHR$(Ø)+RV$+"START OF
   TEXT"+NM$
11Ø TX$(TL)=CHR$(Ø)+RV$+"END OF
   TEXT"+NM$
12Ø CLS:PRINTTAB(13,3)RV$"MAIN
   MENU"NM$TAB(1Ø,6)"(L)OAD"
   TAB(1Ø,8)"(S)AVE"TAB(1Ø,1Ø)
   "(I)/O CHANGE"
13Ø PRINTTAB(1Ø,12)"(E)DIT TEXT"
   TAB(1Ø,14)"(C)LEAR MEMORY"
   TAB(1Ø,16)"(P)RINT OUT"TAB
   (1Ø,18)"(A)LTER PRINTER"
   TAB(1Ø,2Ø)"(Q)UIT PROGRAM"
14Ø B$=GET$
15Ø B=INSTR("LSIECPAQ",B$):IF B=Ø
   THEN 14Ø
16Ø ON B GOSUB 92Ø,8ØØ,11ØØ,27Ø,
   19Ø,148Ø,139Ø,2ØØ
17Ø IF B=5 THEN 23Ø
18Ø GOTO 12Ø
19Ø RETURN
2ØØ CLS:PRINTTAB(13,3)RV$"QUIT
   PROGRAM"NM$:PRINT''"ARE YOU SURE
   (Y/N) ?"
21Ø R$=GET$:IF R$="N" THEN RETURN
22Ø IF R$="Y" THEN CLS:END ELSE 21Ø
23Ø CLS:PRINTTAB(1Ø,5)RV$"CLEAR
   MEMORY"NM$:PRINT''"ARE YOU SURE
   (Y/N) ?"
24Ø B$=GET$:IF B$="N" THEN 12Ø
25Ø IF B$<>"Y" THEN 24Ø
26Ø FOR K=1 TO TL:TX$(K)="":
   NEXT:TL=1:CP=1:GOTO 1ØØ
27Ø CLS:PRINTTAB(13,5)RV$;"EDIT
   MENU"NM$;TAB(1Ø,1Ø)"(T)OP OF
   TEXT"TAB(1Ø,12)"(B)OTTOM OF
   TEXT"TAB(1Ø,14)"(N)EXT LINE OF
   TEXT"TAB(1Ø,16)"(M)AIN MENU"
28Ø B$=GET$
29Ø B=INSTR("TBNM",B$):IF B=Ø THEN
   28Ø
3ØØ CLS
31Ø ON B GOTO 32Ø,33Ø,34Ø,35Ø
32Ø CP=1:GOTO 34Ø
33Ø CP=TL
34Ø GOSUB6ØØ:GOTO36Ø
35Ø RETURN
36Ø A$=""
37Ø P=1
38Ø VDU 23,1,Ø;Ø;Ø;Ø;31,Ø,15:
   PRINTA$"□":VDU 31, (P-1) MOD
   4Ø,15+(P-1) DIV 4Ø,23,1,1;Ø;Ø;Ø;
39Ø TB=GET
4ØØ *FX21,Ø
41Ø IF TB>31 AND TB<127 THEN 5ØØ
42Ø IF TB=13 AND TL<N%-8 THEN
   GOSUB53Ø
```

```
430 IF TB=136 AND P>1 THEN VDU
    8:P=P-1
440 IF TB=137 AND P<=LENA$ THEN
    VDU 9:P=P+1
450 IF TB=Ø OR TB=4 OR TB=19 OR
    TB=135 OR TB=138 OR TB=139 THEN
    SF=Ø:GOSUB 700 ELSE 48Ø
460 IF P>LEN(A$)+1 THEN
    P=LEN(A$)+1
470 IF TB=135 THEN 38Ø ELSE 41Ø
480 IF TB=127 AND P>1 THEN P=P-1:
    A$=LEFT$(A$,P-1)+MID$(A$,P+1)
490 GOTO 38Ø
500 A$=LEFT$(A$,P-1)+CHR$(TB)+
    MID$(A$,P):P=P+1
510 IF LENA$=120 AND TL<N%-8 THEN
    GOSUB 53Ø
520 GOTO 38Ø
530 X=(LENA$ DIV 4Ø)+1+((LEN(A$)
    MOD 4Ø)=Ø)-(LEN(A$)=Ø)
540 FOR K=TL+X TO CP+X STEP
    -1:TX$(K)=TX$(K-X):NEXT
550 IF LENA$>40 THEN TX$(CP)=
    LEFT$(A$,4Ø):A$=MID$(A$,41):CP
    =CP+1:GOTO 55Ø ELSE TX$(CP)=A$
560 CP=CP-X+1
570 A$=" ":P=1
580 TL=TL+X:CP=CP+X:VDU
    23,1,Ø;Ø;Ø;Ø;
590 PRINTTAB(Ø,15)SPC(12Ø)
600 VDU 23,1,Ø;Ø;Ø;Ø;
610 IF CP<5 THEN ST=Ø
    ELSE ST=CP-5
620 VDU3Ø,1Ø:FOR K=ST TO ST+1Ø:
    PRINTTX$(K);SPC(4Ø-LEN(TX$(K))
    -9*(K=Ø OR K=TL));
630 IF K=CP-1 THEN PRINT">";SPC
    (39);
640 NEXT:PRINTSTRING$(4Ø,"#")
650 PRINTTAB(Ø,23)"MEMORY FREE
    =";4Ø*(N%-TL)TAB(2Ø)
    "LINE NO.=";CP"□□□□";
660 VDU 31, (P-1) MOD 4Ø,15+(P-1)
    DIV 4Ø,23,1,1;Ø;Ø;Ø;:RETURN
```

```
10 PRINT"■π";CHR$(8)
20 POKE5328Ø,6:POKE53281,Ø
30 DIMTX$(5Ø1):OW=Ø:EM=Ø:DN=4
35 GC$="■■■■■■■■
   ■■■■■■■■■■■■
   ■■■■"
40 BL$=CHR$(16Ø):TL=1:CP=1:
   MW=8Ø:TW=6Ø:PL=6Ø:TH=4Ø:GP=
   1Ø:LF$=CHR$(13)+CHR$(13)
42 LF$=LF$+CHR$(13):
   GOSUB5ØØØ
50 TX$(Ø)="■■":SW$="":FOR
   F=ØTO39:TX$(Ø)=TX$(Ø)+"+":
   SW$=SW$+"□":NEXTF
55 FORZ=1TO4Ø:SP$=SP$+"□":NEXT
60 TX$(Ø)=TX$(Ø)+"■π":TX$(1)
   ="■■□□□□□□□□□□□□
   .□□□□□□□□□□□□.
   □□□□□□□□□□□□.
   □□□□□□□□□□□□.
   ■π"
70 PRINTCHR$(142)
71 PRINT"■■■"TAB(15);"■MAIN
   MENU■π":PRINT"■■";TAB
   (14);"■L■OAD■"
72 PRINTTAB(14)"■S■AVE■":
   PRINTTAB(14)"■I■/O CHANGE■"
80 PRINTTAB(14)"■E■DIT TEXT■":
   PRINTTAB(14)"■C■LEAR
   MEMORY■"
81 PRINTTAB(14)"■P■RINT OUT■"
82 PRINTTAB(14)"■A■LTER PRINTER
   ■":PRINTTAB(14)"■Q■UIT
   PROGRAM■"
85 PRINTTAB(12)"■■■□□SELECT
   OPTION□□π"
90 GETB$:IFB$=""THEN9Ø
100 B=Ø:FORF=1TO8:IFB$=MID$
    ("LSIECPAQ",F,1)THENB=F
102 NEXTF:IFB=ØTHEN9Ø
110 ON B GOSUB45ØØ,4ØØØ,5ØØØ,
    1ØØØ,16Ø,3ØØØ,55ØØ,13Ø
120 GOTO7Ø
130 PRINTQD$;TAB(11);
    "■■■ARE YOU SURE□(Y/N)?"
140 GET R$:IFR$<>"Y"ANDR$<>"N"
    THEN14Ø
150 IF R$="Y"THENPRINT"■":
    END
155 RETURN
160 PRINT"■■■";TAB(13);"■
    CLEAR MEMORYπ":PRINTTAB
    (1Ø)"■■ARE YOU SURE□(Y/N)?"
170 GETB$:IFB$<>"Y"ANDB$<>"N"
    THEN17Ø
180 IF B$="N"THENRETURN
190 IF TL=1 THEN RETURN
195 FORK=1TOTL-1:TX$(K)="":NEXT:
    TX$(1)=TX$(TL):TX$(TL)="":
    TL=1:CP=1:RETURN
1000 PRINT "■N■■";TAB(15);
     "■■DIT■EXTπ":PRINT
     "■■";TAB(14);"■■■
     OP OF TEXT"
1002 PRINTTAB(14)"■■■■
     OTTOM OF TEXT":PRINTTAB(14)
     "■■■■EXT LINE OF TEXT"
1004 PRINTTAB(14)"■■■■AIN MENU"
1010 GETB$:IFB$=""THEN1Ø1Ø
1020 B=Ø:FORF=1TO4:IFB$=MID$
     ("TBNM",F,1)THENB=F:GOTO 1Ø3Ø
1022 NEXTF:GOTO 1Ø1Ø
1030 ON B GOTO1Ø5Ø,
     1Ø6Ø,1Ø7Ø 1Ø8Ø
1040 IFTL=<3ORCP=1THEN1Ø7Ø
1050 CP=1:GOTO1Ø7Ø
1060 CP=TL
1070 PRINT"■":GOSUB2Ø9Ø:GOSUB
     15ØØ:GOTO1ØØØ
1080 RETURN
1500 A$="□":P=Ø
1505 IFEM=1THEN25ØØ
1510 POKE 198,Ø
1515 PRINT LEFT$(GC$,23)A$;
1520 CH=PEEK(19Ø4+P):GETT$
1521 IFLEN(A$)=81 OR T$=CHR$(13)
     THEN GOSUB2ØØØ:GOTO 1515
1522 CH=(CH+128)AND255:POKE19Ø4
     +P,CH:CH=(CH+128)AND255:POKE
     19Ø4+P,CH
1530 IFT$=""ORT$=CHR$(34)THEN
     152Ø
1540 IFT$="■"ORT$="■"THEN
     152Ø
1550 IFP<LEN(A$)-1ANDT$=CHR$(1Ø)
     THENA$=LEFT$(A$,P)+MID$(A$,
     P+2):GOTO16ØØ
1551 IFT$=CHR$(1Ø)THEN152Ø
1555 IFT$=CHR$(136)THENRETURN
1560 IFT$=CHR$(134)ANDCP=1THEN152Ø
1561 IFT$=CHR$(134)THEN
     A$=TX$(CP-1)+"□":T$="":
     P=Ø:GOSUB2Ø9Ø:GOTO151Ø
1562 IFT$=CHR$(135)THEN OW=ABS
     (1-OW):GOSUB 2Ø9Ø:GOTO 151Ø
1563 IFT$="■"THENCP=1:
     GOSUB2Ø9Ø
1564 IFT$=CHR$(133)THENEM=1:
     PM=CP:GOSUB2Ø9Ø:GOTO15Ø5
1565 IFT$="■"THENGOSUB2Ø9Ø:
     GOTO15ØØ
1567 IFT$<>CHR$(2Ø)THEN158Ø
1570 IF P=Ø THEN 151Ø
1572 A$=LEFT$(A$,P-1)+MID$(A$,
     P+1):P=P-1
1575 PRINT LEFT$(GC$,23);:
     FORF=1TOLEN(A$)-2:PRINT
     "■";:NEXT
1577 PRINT"□□":GOTO151Ø
1580 KJ=Ø:IFT$=CHR$(148)THEN
     T$="□":KJ=1
1582 IFT$<>CHR$(157)ANDT$<>CHR$
     (29)ANDASC(T$)<32THEN151Ø
1590 IF T$=CHR$(29)ORT$=CHR$
     (157)THEN161Ø
1591 IFOW=1ANDJ<>1THEN1595
1593 A$=LEFT$(A$,P)+T$+MID$
     (A$,P+1):P=P+1:GOTO16ØØ
1595 B$=A$:A$=LEFT$(A$,P)+T$+
     MID$(A$,P+2)
1598 P=P+1:A$=A$+"□"
1600 PRINTLEFT$(GC$,23)A$;
1610 IFT$=CHR$(29)ANDP<LEN(A$)-1
     THENP=P+1
1620 IF(T$=CHR$(157)ANDP>Ø)OR
     KJ=1THENP=P-1
1630 GOTO152Ø
```

CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT.*

A

Applications
text-editor program 852–856
ASCII codes
of f keys
Acorn 829
Commodore 64, Vic 20 826
Assets in games 830–837
ATTR
adding a new instruction
Spectrum 844–847

B

BASIC
adding instructions to
Acorn, Dragon, Spectrum 844–851
Basic programming
designing a new typeface 838–843
programming function keys 825–829
Business strategy game
see Goldmine
BYE
adding a new instruction
Acorn 847–849

C

Command strings
use of with f keys
Commodore 64, Vic 20 826–828
Control codes
of f keys
Commodore 64, Vic 20 826

D

DATA statements
use of for custom typeface 838–843
Datafiles
use of in text editor 852
Drawing a new typeface 838–843

E

Editing
using f keys
Acorn 829
Also see text-editor program
Enlarging a typeface 838–843

F

Function keys, programming
advantages of 826
Acorn 828–829
Commodore 64, Vic 20 826–828
FX command
use of with f keys
Acorn 829

G

Games
goldmine 830–837
Goldmine game
part 1—basic routines 830–837
Graphics
hi-res for custom typeface 838–843
in goldmine game 832–837

I

INKEY
use of to detect keypresses
Acorn 829
Input/output options
in text editor 852
Instructions, adding to BASIC
Acorn, Dragon, Spectrum 844–851
Interface 1
Spectrum 846
INV
adding a new instruction
Acorn 847–849
INVERSE
adding a new instruction
Spectrum 844–847
INVERT
adding a new instruction
Dragon 849–851

K

Keypresses
detecting
Acorn 829
Commodore 64, Vic 20 827
how they work 826
Keys, function
Acorn, Commodore 64, Vic 20 825–829

L

Letter-generator program
Acorn, Commodore 64, Dragon, Spectrum, Tandy 838–843
LINE
use of to design typeface
Dragon, Tandy 840–843
☆ LINE
Acorn 847–849
LIST
with f keys
Acorn 829
Commodore 64, Vic 20 827
Loader program, use of
Acorn, Dragon, Spectrum, Tandy 842–843
LOADing
a custom typeface
Acorn, Dragon, Spectrum, Tandy 842–843

M

Machine code
routine to add to BASIC
Acorn 847–848
Dragon 849
Spectrum 844
Memory
storing new keystrokes in
Acorn 829
Commodore 64, Vic 20 827–828
storing new typeface in
Acorn, Commodore 64, Dragon, Spectrum, Tandy 842
Menus
in text-editor program 852–853

O

OLD
adding a new instruction
Dragon 849–851
Operating system software
Acorn 828
Commodore 64, Vic 20 826

P

PLOT
use of to design typeface
Acorn, Spectrum *838–843*
Pointers
re-setting with OLD
Dragon 849–851
POKE
use of to design typeface
Commodore 64 839–842

R

RETURN
use of with f keys
Acorn 828–829
Commodore 64, Vic 20 827–828
RND function
in goldmine game 832–837
ROM
switching between old and new
Spectrum 846–847

S

SAVEing
custom typeface
Acorn, Commodore 64, Dragon, Spectrum, Tandy 842–843
Scaling a custom typeface 841–843
Screen display
SAVEing and LOADing
Acorn, Dragon, Spectrum, Tandy 842–843
Strings
in custom typeface program 841
use of with f keys
Acorn 828
Commodore 64, Vic 20 826–827
Stubs
Dragon 849–850

T

Text-editor program
Acorn, Commodore 64, Dragon, Spectrum
part 1 852–856
Tokens,
recognizing new 844–851
Typeface. setting up new
Acorn, Commodore 64, Dragon, Spectrum, Tandy 838–843
Typing
speeding up using f keys
Acorn, Commodore 64, Vic 20 825–829

U

Utility packages
Commodore 64, Vic 20 *827*

V

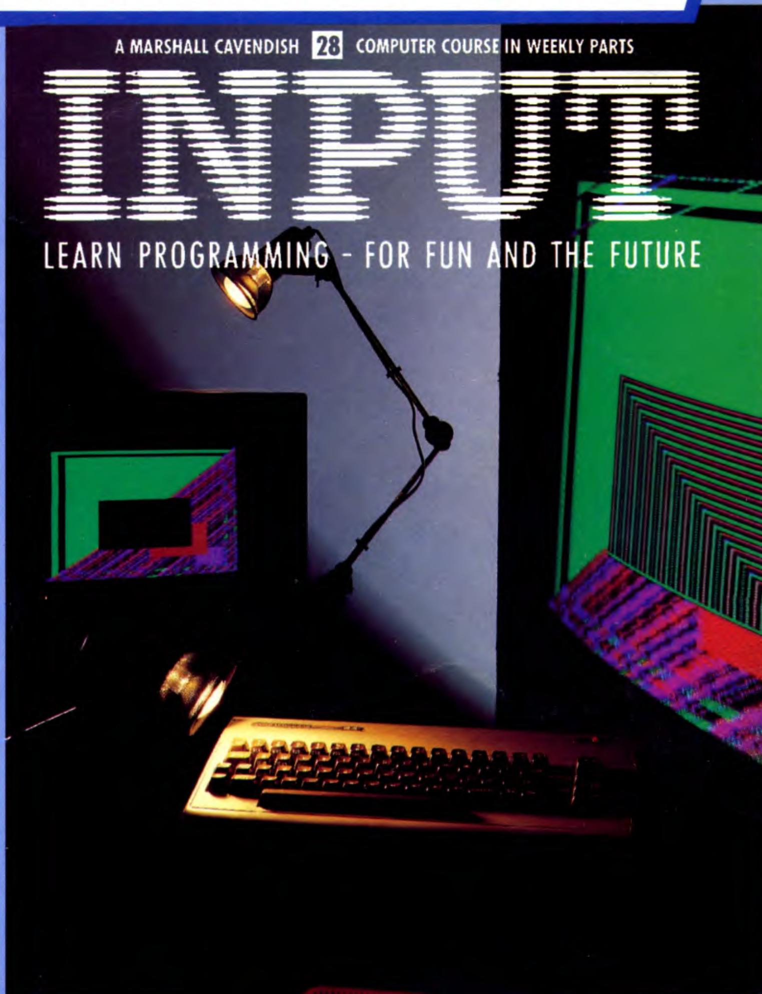
VECTORS, redirecting 844–851

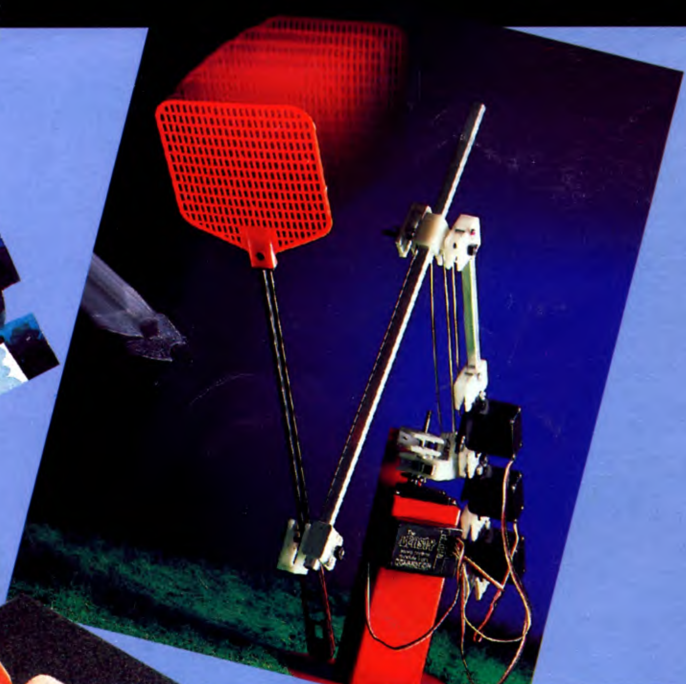
W

Work area
of text-editor 853

COMING IN ISSUE 28...

- *Slice into the world of CONIC SECTIONS, and lick the problems of drawing this fascinating family of curves on your Micro*
- *It's off to the Gold Exchange for successful tycoons in part two of GOLDMINE*
- *Arm yourself with a ROBOT. Control these exciting devices as the Age of Robotics dawns*
- *Continue with part two of the TEXT EDITOR. Find out about the editing functions*
- *Plus ... for Commodore 64 users, extending the graphics capabilities of the HI-RES UTILITY*





ASK YOUR NEWSAGENT FOR INPUT